

DOCUMENT RESUME

ED 057 579

EM 009 419

AUTHOR Feurzeig, Wallace; And Others
 TITLE An Introductory LOGO Teaching Sequence: LOGO Teaching Sequence on Logic, LOGO Reference Manual.
 INSTITUTION Bolt, Beranek and Newman, Inc., Cambridge, Mass.
 SPONS AGENCY National Science Foundation, Washington, D.C.
 REPORT NO R-2165
 PUB DATE 30 Jun 71
 NOTE 135p.; Programming-Languages as a Conceptual Framework for Teaching Mathematics, Volume One; See also EM 009 420, EM 009 421, EM 009 422

EDRS PRICE MF-\$0.65 HC-\$6.58
 DESCRIPTORS *Computer Assisted Instruction; Geometry; Logic; Manuals; *Mathematics Instruction; *Programing Languages
 IDENTIFIERS Project LOGO

ABSTRACT

In earlier work a programing language, LOGO, was developed to teach mathematics in the framework of computer programs. Using LOGO a few programs were tested in both elementary and junior high school mathematics classrooms with excellent results. The work reported here is the first effort to systematically develop extensive curriculum materials using the LOGO language. This first volume gives a reference manual on the LOGO language and two of the LOGO teaching sequences. The short introductory sequence, on teletype geometry, is one of many possible starting points for studying LOGO. The sequence on logic is the most advanced of those presented; it has a sophisticated ability for dealing with syllogistic-type arguments. For subsequent volumes see EM 009 420, EM 009 421, and EM 009 422.
 (JY)

B O L T B E R A N E K A N D N E W M A N I N C

C O N S U L T I N G • D E V E L O P M E N T • R E S E A R C H

Report No. 2165

Volume 1

PROGRAMMING-LANGUAGES AS A CONCEPTUAL
FRAMEWORK FOR TEACHING MATHEMATICS

An Introductory LOGO Teaching Sequence

LOGO Teaching Sequence on Logic

LOGO Reference Manual

U.S. DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
OFFICE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRO-
DUCED EXACTLY AS RECEIVED FROM
THE PERSON OR ORGANIZATION ORIG-
INATING IT. POINTS OF VIEW OR OPIN-
IONS STATED DO NOT NECESSARILY
REPRESENT OFFICIAL OFFICE OF EDU-
CATION POSITION OR POLICY.

Submitted to:

National Science Foundation
Office of Computing Activities
1800 G Street, N.W.
Washington, D. C. 20550

Contract NSF-C 615

30 June 1971

1

C A M B R I D G E

N E W Y O R K

C H I C A G O

L O S A N G E L E S

ED057579

EM 009 419

ED057579

PROGRAMMING-LANGUAGES AS A CONCEPTUAL
FRAMEWORK FOR TEACHING MATHEMATICS

Final Report on the second fifteen
months of the LOGO Project

Wallace Feurzeig
George Lukas
Philip Faflick
Richard Grant
Joan D. Lukas
Charles R. Morgan
Walter B. Weiner
Paul M. Wexelblat

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

30 June 1971

Submitted to:
National Science Foundation
Office of Computing Activities
1800 G Street, NW
Washington, D. C. 20550

Contract NSF-C 615

CONTENTS

Volume 1

An Introductory LOGO Teaching Sequence

LOGO Teaching Sequence on Logic

LOGO Reference Manual

Volume 2

LOGO Teaching Sequences on Numbers
and

Functions and Equations

Volume 3

LOGO Teaching Sequences on
Strategy in Problem-Solving
and

Story Problems in Algebra

Volume 4

The LOGO Processor

A Guide for System Programmers

PREFACE

This is the final report of work under Contract NSF-C 615, "Programming-Languages as a Conceptual Framework for Teaching Mathematics." In earlier directly-related work, supported by the Office of Computing Activities, we advanced the thesis that mathematics could be developed and presented in the framework of programs, and that this kind of presentation would greatly enhance teaching and learning. Using the LOGO programming language designed expressly for this purpose, we tested this thesis in both elementary and junior high school mathematics classrooms, obtaining the excellent results we had anticipated.*

The teaching materials used in this earlier work were developed "on the run" to meet immediate class needs. These materials were, therefore, unconnected or loosely connected segments of partially realized designs. The focus of this phase of the work was on classroom experimentation and testing, not on curriculum development. It was clear to us from the outset, however, that extensive curriculum material would eventually be needed. The object of the work reported here is to take the first steps in the development of such materials.

For this purpose, we have developed five extended LOGO teaching sequences and an introductory LOGO sequence. These range over a variety of elementary mathematical subjects, levels of difficulty, and mathematical and pedagogic style. This diversity is intentional. We want to illustrate the scope of this new approach to mathematics and its general accessibility to teachers and students.

*"Programming-Languages as a Conceptual Framework for Teaching Mathematics," Final Report on the first fifteen months of the LOGO Project, Wallace Feurzeig et al, Nov. 1969.

The report is composed of four volumes. Volume 1 contains a reference manual on the LOGO language and two of the LOGO teaching sequences. The short introductory sequence, on teletype geometry, is one of many good starting points for studying LOGO. The LOGO sequence on logic is by far the most advanced of those presented here. In it we develop a sophisticated ability for dealing with syllogistic-type arguments. The sequence demonstrates that an extremely complex set of procedures can be evolved in a natural way.

Volume 2 includes two LOGO teaching sequences treating some standard mathematical topics -- on the representation of numbers and the algorithms of arithmetic, and on functions and equations -- but incorporating the new viewpoints made possible by the use of LOGO programs. The sequence on numbers is the most detailed presentation of the series in accordance with its mathematical content. Their content and level of difficulty makes these two sequences well-suited for incorporation into existing curricula.

Volume 3 is comprised of two LOGO teaching sequences on "problem-solving". One deals with the generation and testing of methods and strategies. The other treats the problem of translating between different representations, both formal and informal. The first sequence uses a variety of mathematical contexts; the second uses the context of story problems in algebra.

We do not regard the teaching sequences as literal teaching prescriptions. Rather, we designed them as exemplary materials to acquaint prospective teachers with the rich mathematical and pedagogical possibilities inherent in the use of LOGO. The sequences are intended as source materials for assisting teachers in the preparation of courses. We think the presentations will

be useful in providing teachers general guidelines around which to make their own variations and extensions.

Volume 4 of this report stands apart from the other three. It contains a detailed description of the LOGO processor algorithm. It was written for system programmers and others interested in the details of LOGO's inner workings. It will facilitate the implementation of LOGO on computer systems of many kinds.

The main authors of the material in this report are Wallace Feurzeig, George Lukas, and Richard Grant. Philip Faflick made major contributions to the number and strategy sequences. Joan Lukas is a co-author of the logic sequence. Charles R. Morgan contributed both to the curriculum material and the LOGO system programming and documentation. Primary work in the latter area was done by Walter B. Weiner with the assistance of Paul M. Wexelblat. The demanding technical typing and drawing tasks were directed and performed by Pearl Stockwell.

Volume 1, Part 1

AN INTRODUCTORY LOGO TEACHING SEQUENCE

The LOGO Project

NSF-C 615

Wallace Feurzeig

George Lukas

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

AN INTRODUCTORY LOGO TEACHING SEQUENCE

We introduce students to LOGO by developing some procedures for making pictures, specifically "geometric" figures, on a teletypewriter. To begin with, we write procedures like the following, for drawing a rectangle:

```
TO RECTANGLE
1 PRINT "++++"
2 PRINT "++++"
3 PRINT "++++"
END
```

To use this procedure, we simply type

RECTANGLE

(When we use a procedure we will underscore our typing to distinguish it from the computer's response)

and LOGO responds:

```
++++
++++
++++
```

But this kind of procedure is pointillistic -- the figure is being specified point by point. To improve upon this, we can write procedures for drawing line segments of specified length to serve as building blocks for drawing a number of different kinds of figures. For example, the following procedures:

```
TO 1+
1 PRINT "+"
END
```

```
TO 3+
1 PRINT "+++"
END
```

```
TO 2+
1 PRINT "++"
END
```

```
TO 4+
1 PRINT "++++"
END
```

type segments of a single + mark, 2 + marks, 3 + marks, and 4 + marks, respectively.

These procedures are immediately used as parts of other, more interesting procedures for drawing figures of fixed shape and size. For example, we can describe a two-by-four rectangle,

```
TO RECTANGLE
1 4+
2 4+
END
```

To draw the rectangle, we type:

```
RECTANGLE
++++
++++
```

To make a larger rectangle we write:

```
TO BIG-RECTANGLE
1 4+                (draw a row)
2 BIG-RECTANGLE     (and repeat the process)
END
```

This procedure will go on indefinitely:

BIG-RECTANGLE

++++
++++
++++
++++
++++
++++
++++
++++
...
...
...

until we forcibly stop it, using the teletype "break" key.

To make a rectangle of specified size, say /N/ rows, we rewrite BIG-RECTANGLE as follows:

```
TO BIG-RECTANGLE /N/
1 TEST IS /N/ 0 (If all rows are done)
2 IF TRUE STOP (Stop)
3 4+ (Otherwise, draw a row)
4 BIG-RECTANGLE (DIFF OF /N/ AND 1) (and repeat the process
END /N/-1 more times)
```

This allows us to draw rectangles with any desired number of rows, for example:

BIG-RECTANGLE 5

++++
++++
++++
++++
++++

We can also write procedures for drawing triangles:

TO TRIANGLE

1 1+

2 2+

3 3+

4 4+

END

TRIANGLE

+

++

+++

++++

But we run into difficulty when we try to extend this to make arbitrarily large triangles. We can accomplish this, however, by writing a single procedure MARK /N/ to type out any specified number /N/ of + marks. By varying the input of this new procedure, we can draw triangles as well as many other regular figures.

In fact, we will write two drawing procedures, BLANK /N/ and MARK /N/. The first of these types out /N/ blank spaces; the second types /N/ +'s and returns the carriage. These two procedures are virtually identical in form. Thus, for example:

TO MARK /N/

1 TEST IS /N/ 0

2 IF TRUE SKIP

3 IF TRUE STOP

4 TYPE "+"

5 MARK (DIFFERENCE OF /N/ AND 1) (and repeat the process /N/-1 more times)

END

(If there are no more marks to type)

(skip to the next line)

(and stop)

(Otherwise, type a mark)

MARK 17

+++++

To illustrate the use of these basic procedures, let's write a procedure to make a little triangle, indented from the left margin.

TO MAKE-A-TRIANGLE

```
1 BLANK 10
2 MARK 1
3 BLANK 9
4 MARK 3
5 BLANK 8
6 MARK 5
END
```

MAKE-A-TRIANGLE

```
  +
  +++
+++++
```

This would be a tedious way of describing larger figures. A considerable improvement comes about from noting that in many cases we wish to center all rows of marks with respect to the same interval. We can easily write a procedure to do this. It will type $/N/$ marks centered in an interval of length $/L/$. The number of spaces it needs to indent before typing is given by the quotient of $(/L/ - /N/)$ and 2. (We must compute this to the nearest integer since we cannot half-space on the teletype.) In LOGO this is expressed QUOTIENT OF (DIFFERENCE OF $/L/$ AND $/N/$) AND 2.

Now we can write the procedure MIDDLE $/N/$, for typing +'s in the middle $/N/$ spaces of an interval of length $/L/$.

TO MIDDLE $/N/$

```
1 BLANK QUOTIENT OF (DIFFERENCE OF  $/L/$  AND  $/N/$ ) AND 2
2 MARK  $/N/$ 
END
```

Using MIDDLE we can now write MAKE-A-TRIANGLE without worrying about spacing.

```

TO MAKE-A-TRIANGLE
1 MIDDLE 1
2 MIDDLE 3
3 MIDDLE 5
END

```

MIDDLE has immediate and broad application as a basic procedure for drawing symmetric figures. To show its utility, let's first define a procedure for typing a rectangle with /A/ columns and /B/ rows.

```

TO RECTANGLE /A/ /B/
1 TEST IS /B/ 0 (If /B/ has become 0,
2 IF TRUE STOP stop)
3 MIDDLE /A/ (Else type /A/ +'s)
4 RECTANGLE /A/ (DIFFERENCE OF /B/ AND 1) (and repeat the process
END /B/-1 times)

```

This procedure executes the command MIDDLE /A/ (which types /A/ "centered" +'s) /B/ times.

RECTANGLE 18 3

```

+++++
+++++
+++++

```

Rectangles are useful as basic building blocks for composing other figures. Trapezoids are also useful, particularly for building many kinds of polygons. And, they include triangles as a limiting case.

```

TO TRAPEZOID /A/ /B/ (/A/ and /B/ are the lengths of the two bases)
1 MIDDLE /A/ (Type /A/ centered +'s)
2 TEST IS /A/ /B/ (If /A/ has become equal to /B/,
3 IF TRUE STOP stop)
4 TRAPEZOID (SUM OF /A/ AND 2) /B/ (Otherwise, repeat the process
END with /A/ increased by 2)

```

(Note that this procedure closely parallels that for RECTANGLE.)

TRAPEZOID 3 9

```
+++
+++++
+++++++
+++++++++
```

TRIANGLE is the limiting case of TRAPEZOID, with $/A/ = 1$.

```
TO TRIANGLE /N/
1 TRAPEZOID 1 /N/
END
```

TRIANGLE 7

```

+
+++
+++++
++++++++
```

Though the TRAPEZOID procedure can be used to draw trapezoids of different sizes, it can only draw trapezoids with the same interior angles, because successive rows increase in width from a smaller to a larger base by a fixed step. TRAPEZOID can be generalized in a very straightforward way to incorporate a larger class of trapezoids. All we need do is include the step-size increment $/STEP/$ as an input. Doing this, the definition becomes,

```
TO TRAPEZOID /A/ /B/ /STEP/
1 MIDDLE /A/
2 TEST IS /A/ /B/
3 IF TRUE STOP
4 TRAPEZOID (SUM OF /A/ AND /STEP/) /B/ /STEP/
END
```

(The only changes have been the inclusion of $/STEP/$ in the title line and in line 4.)

Now we can draw trapezoids with relatively big slopes:

TRAPEZOID 4 28 6

```
      +++++
    ++++++
  ++++++
+++++
+++++
```

We can also draw "upside-down" trapezoids by using negative increments.

TRAPEZOID 13 5 -2

```
+++++
+++++
+++++
+++++
+++++
```

It is possible to specify values of /A/, /B/, and /STEP/ that do not properly define a trapezoid. For example, if the top base is to be 8 and the bottom base is to be 3, with step size +1, the drawing process will never terminate by itself. The procedure TRAPEZOID can easily be modified to check for this and all other nonterminating cases. Discovering and fixing such difficulties provides good problems for the students as a natural side-effect of their own work.

The power of the procedures we have just defined is evident from the ease with which we can use them to draw a large variety of other figures. Thus, a HEXAGON can be built from two trapezoids; a DIAMOND from two triangles; a PENTAGON from a trapezoid and a triangle; an OCTAGON from a trapezoid, a rectangle, and another trapezoid; and so on. For example, this is a procedure for drawing hexagons, where /A/ is the starting width and /B/ is the mid-width.

```
TO HEXAGON /A/ /B/ /STEP/
1 TRAPEZOID /A/ /B/ /STEP/
2 TRAPEZOID (DIFFERENCE OF /B/ AND /STEP/) /A/ (-/STEP/)
END
```

Here are examples of figures made by such procedures.

HEXAGON 4 16 4

```

      +++++
    +++++++
  ++++++++
+++++
+++++
+++++
+++++

```

PENTAGON 5 9 2

```

      +++++
    +++++
  +++++
  +++++
  +++++
  +++++
  +

```

DIAMOND 7 2

```

      +
    +
  +
  +
  +
  +
  +
  +
  +

```

OCTAGON 4 12 4

```

      +++++
    +++++
  +++++
  +++++
  +++++
  +++++
  +++++
  +++++

```

These figure-drawing procedures can themselves be used as construction elements. For example, we can stack figures together to form towers such as the following one formed by hexagons of increasing size.

HEX-TOWER 4 8 3

```

      + + +
    + + + + +
  + + + + + + +
    + + + + +
      + + + +
        + + + + + + +
      + + + + + + + + +
    + + + + + + + + + + +
  + + + + + + + + + + + +
+ + + + + + + + + + + + +
  + + + + + + + + + + + +
    + + + + + + + + + + +
      + + + + + + + + +
        + + + + + + + +
          + + + + + + +
            + + + + + +
              + + + + +
                + + + +
                  + + +
                    + +
                      +

```

The LOGO procedure for generating this kind of tower follows.
(Note how similar it is in form to many previous ones.)

TO HEX-TOWER /A/ /B/ /NUM/

(/A/ and /B/ define the topmost hexagon, /NUM/ is the number of hexagons in the tower)

1 TEST IS /NUM/ Ø

```
2 IF TRUE STOP
```

3 HEXAGON /A/ /B/ 2

(Draw a hexagon of dimensions
/A/, /B/, 2)

4 HEX-TOWER (PRODUCT OF 2 AND /A/)
(PRODUCT OF 2 AND /B/)
(DIFFERENCE OF /NUM/ AND 1)

```
(Increase the hexagon size and
repeat the process (/NUM/ - 1)
times)
```

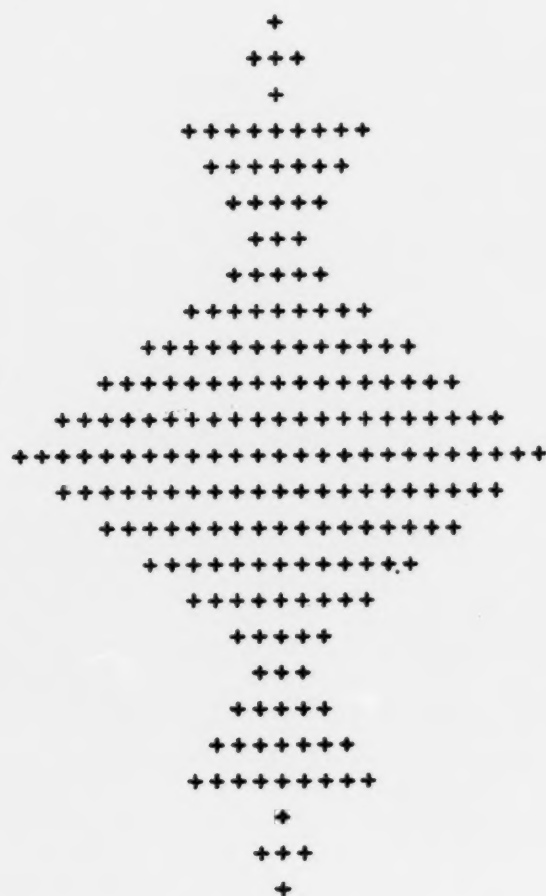
END

An interesting way of generating more complex symmetric structures is by using a random process to determine the constituent figures and their sequencing. We can develop a LOGO program, PATTERN /NAME/ /NUMBER/, to do this. PATTERN creates a drawing program, whose name is /NAME/, which uses /NUMBER/ figures to create a pattern that is symmetric about the horizontal as well as the vertical. Here, for example, are two especially nice LOGO procedures created by PATTERN, along with their resulting drawings.

PATTERN "TOTEM" 5

```
TO TOTEM
1 DIAMOND 3 2
2 TRAPEZOID 9 3 -2
3 HEXAGON 5 25 4
4 TRAPEZOID 3 9 2
5 DIAMOND 3 2
END
```

TOTEM



PATTERN "FIGURE-12" 3

TO FIGURE-12
1 TRIANGLE 11 2
2 HEXAGON 15 31 4
3 TRIANGLE 11 -2
END

FIGURE-12



Using the procedures written thus far, the student can generate a large number of different geometric shapes, and the writing of such a sequence of procedures represents a considerable achievement on his part. The foundations of geometry, however, lie in the transformation of geometric objects, not merely in their portrayal. Our next goal, then, is to write procedures for performing standard transformations of geometric figures. Such transformations include translation, rotation, and reflection, both with respect to a given point and with respect to a given

line. Procedures to generate the union and intersection of the sets of points defining two geometric objects are also useful. And we need a procedure DRAW which plots any given set of points.

To write these procedures we need a different representation for geometric objects, one which can be retained within the computer. (Clearly we do not have such a representation thus far -- our current objects are generated and drawn one line at a time.) Perhaps the simplest such representation is a list of pairs of numbers, each pair representing one point of the object. Then it is easy to write procedures, such as the following, which reflect a set of points about the x-axis.

TO REFLECTX /PAIR LIST/	(/PAIR LIST/ is the list of X Y number pairs)
1 TEST IS /PAIR LIST/ /EMPTY/	(Are there any points left on /PAIR LIST/? If not, terminate procedure)
2 IF TRUE OUTPUT /EMPTY/	
3 OUTPUT LIST OF	
FIRST OF /PAIR LIST/	
NEGATIVE OF SECOND OF /PAIR LIST/	
REFLECTX OF (BUTFIRST2 OF /PAIR LIST/)	(Otherwise, output a list of the X coordinate and the negative of the Y coordinate of the first number pair on /PAIR LIST/, and REFLECTX applied to the pair list obtained by deleting the first number pair on /PAIR LIST/)
END	

Thus for example:

```
PRINT REFLECTX OF "1 2 4 3 7 11 2 -1"
1 -2 4 -3 7 -11 2 1
```

Using this and two similar procedures, one for reflecting about the 45 degree line through the origin, and the other for reflecting about the Y-axis, we can now write a procedure for random generation of figures having eightfold symmetry.

Volume 1, Part 2

LOGO TEACHING SEQUENCE ON LOGIC

Teacher's Text

and

Problems

The LOGO Project

NSF-C 615

Joan D. Lukas

George Lukas

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

CONTENTS

	Page
0. Introduction	i
1. Formal Background	1
2. Generalization of Lewis Carroll Diagrams	18
3. Generating Logic "Diagrams"	20
4. Marking the "Diagrams"	28
5. Diagrams and Premisses	36
6. Testing Syllogisms	40
7. More General Statement Forms	47
8. Generalizing Our Syllogism Tester	49

Problems

LOGO UNIT ON LOGIC

0. Introduction

Syllogisms formed the central core of logic until they were pushed aside by the development of mathematical logic in the Nineteenth Century. We have chosen to focus on them here because they are accessible without a great deal of formalism and because syllogistic types of argument occur frequently in everyday discourse.

This section develops a set of LOGO procedures which test the validity of syllogistic arguments. LOGO is peculiarly suited for such an application because of its non-numerical capabilities and its procedure-oriented programming heuristic. Algorithms in everyday language are first developed for each part of the rather extensive syllogism tester; the translation of these into LOGO procedures follows in a very natural manner. The method of testing syllogisms used is an adaptation of the one presented by Lewis Carroll in his book "Symbolic Logic".

The ideas underlying this section and the related background in symbolic logic were provided by Joan Lukas. George Lukas implemented this treatment in the form of LOGO procedures. Both shared in the actual writing.

1. Formal Background

Many of the inferences made in both formal and informal reasoning concern the relations among classes of objects. We may say, for instance, that *computers* can't *think* since *computers* are *machines* and *machines* can't *think*. While one may dispute one or the other of the premises, the structure of this argument is unassailable:

- (1) All A are B
- (2) All B are C
- (3) $\therefore$ All A are C

Once one has accepted (1) and (2), (3) inevitably follows.

If we had made a different statement about the classes, say that they overlap rather than that one is contained in the other, the argument is changed radically. We may not conclude that some babies are fifty years old if we know that some babies are male and some males are fifty years old. Here the structure of the argument is

- (4) Some A are B
- (5) Some B are C
- (6) $\therefore$ Some A are C

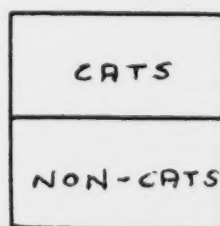
In this case (6) does not follow from (4) and (5).

Statements which express relations between classes, such as "All computers are machines," and "Some babies are male," are known as categorical statements. Arguments such as (1) (2) - (3) and (4) (5) - (6), whose premises and conclusions are categorical statements, are (categorical) syllogisms. The study of such statements and arguments and in particular the singling out of valid forms of syllogisms is a central concern of classical logic.

Categorical statements were first studied systematically by Aristotle who recognized four types of such statements. One of these asserts the containment of one class in another -- "All cats are animals" or "It always rains on Tuesday" (which can be translated as "Every Tuesday is a day when it rains"). By negating such a statement, we obtain a second type "Not all cats are animals" or "It doesn't always rain on Tuesday".

A third type of statement asserts that two classes have nothing in common: "No cats are dogs" "It never snows in August." Note that this statement is not equivalent to the statement above, i.e., "It never rains on Tuesday" is a stronger assertion than "It doesn't always rain on Tuesday." The fourth type of statement asserts that two classes do have something in common: "Some cats like milk" (or "*Some cats are things that like milk*").

The meaning of these types of statements, and the relations among them, can be made clearer by a diagrammatic representation. Consider the statement "Some cats are animals". We can imagine the universe partitioned into two classes -- cats and noncats.



The universe can also be divided into animals and nonanimals.



Lewis Carroll devised a means of representing the four possible classes obtained by combining these two partitions

CATS AND ANIMALS	CATS AND NON- ANIMALS
NON- CATS AND ANIMALS	NON- CATS AND NON- ANIMALS

and using markers to indicate the existence or nonexistence of members of each class, as follows. An x in a region indicates that the condition represented by that region is never satisfied, while a • indicates that there are some objects (at least one object) which fall into the class. Thus, in the diagram above, an x in the upper right region indicates

	ANIMALS	NON-ANIMALS
CATS		X
NON-CATS		

that there are no cats which are not animals and a • in the upper left region indicates that there are cats which are animals.

•	

Similar diagrams can be used for any two classes A and B. If the right-hand side represents A, the left not A, the upper-half B, and the lower not B, we get

A, B	A, not B
not A, B	not A, not B

Then, "Some A are B" can be represented by

●	

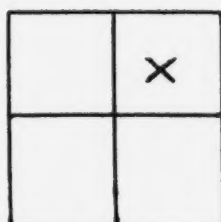
"Some A are not B" by

	●

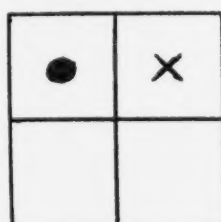
"No A are B" is represented by

×	

In drawing the diagram corresponding to the statement "All A are B", we have to make a decision. It is clear that, if all A are B then no A are not B. So the diagram should have an x in the upper right region.



Is this the complete diagram for the statement, or does it also imply that there are some A's which are B's, so that a • would appear in the upper left region?

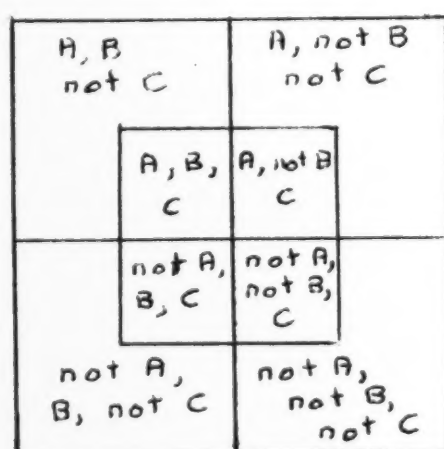


The circle seems to be appropriate because, if all A's are B's, than certainly some A's are B's. A difficulty arises, however, when we consider the possibility that there might not be any A's at all. Then "All A are B" might be taken as vacuously true. But, "Some A are B" would be false, since it implies the existence of A's.

Consider, for example, the statement "All round squares are pink." In most circumstances we would dismiss this statement as nonsense on the ground that there are no round squares and so there is no point in saying anything about them. If we were forced to assign a truth value to the statement, we might interpret it to mean both that round squares exist and every one of them is pink. Under this interpretation, the statement is false. Alternatively, we might rule it to be vacuously true, since it asserts that the class of round squares is contained in the class of pink objects and since the class of round squares is empty, it is contained in every class.

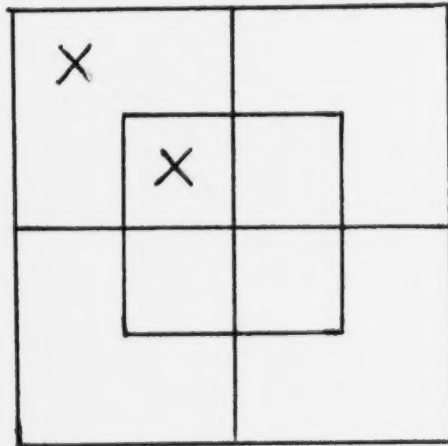
In mathematics a statement of the form "All A are B" is usually not taken to imply the existence of A's. We can talk meaningfully about the class of counter examples to Fermat's last theorem* without knowing whether any such counter examples exist. In ordinary language, however, this view can lead to absurdity. For instance, we would not want to accept the statement "All pink elephants are two feet tall" as true merely on the ground that there are no pink elephants. So we will take the diagram for "All A are B" to contain both an x in the A, not B region and a • in the A,B region, that is, the statement is taken as implying both "Some A are B" and "No A are not B".

In the study of categorical syllogisms, we are concerned with three classes of objects rather than two. The Lewis Carroll diagram is extended to three classes by dividing it into an inner square and an outer region, the inner square representing the third class, C, and the outer region the complement of this class, not C.

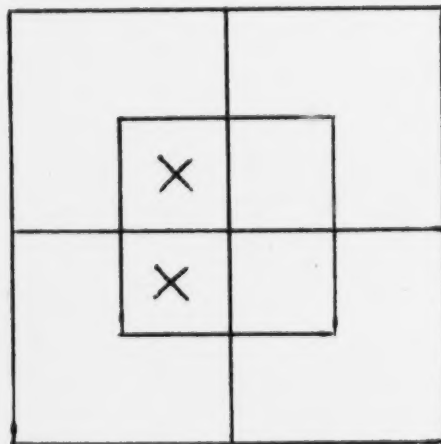


* Fermat's last theorem asserts that there are no integers x, y, z, and n with $n > 2$ satisfying the equation $x^n + y^n = z^n$.

The diagram corresponding to the statement "No A are B" now becomes

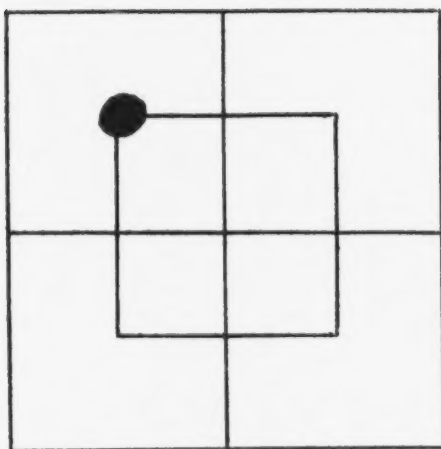


and "No B are C" has the diagram

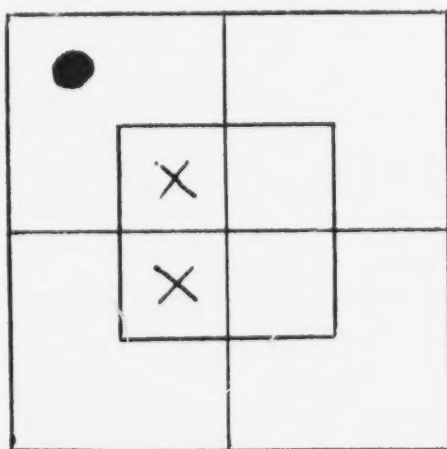


Clearly, two regions are excluded in each case.

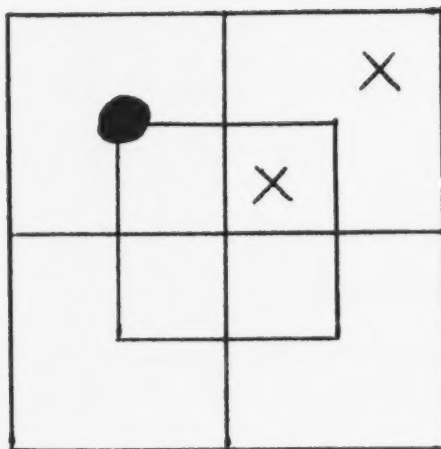
The statement "Some A are B" is now more difficult to diagram. It does not tell us whether the A-B objects belong to C or to its complement. We indicate this uncertainty by placing the circle on the line dividing the two regions A,B,C, and A,B,not C. It is "sitting on the fence".



Further statements may tell us which region the circle belongs in. For instance, "No B are C" would eliminate the inner region and force the circle into the upper left corner. The diagram resulting from these two statements is



We have interpreted "All A are B" to mean both "No A are not B" and "Some A are B" so the corresponding diagram is



We can now use these diagrams to check the validity of syllogistic arguments. A syllogism is valid if the premises imply the conclusion. In terms of diagrams, this means the diagram obtained by combining the premises contains x's and •'s wherever the conclusion's diagram does. The first syllogism we discussed was

- (1) All computers are machines
- (2) Machines can't think
- (3) $\therefore$ Computers can't think

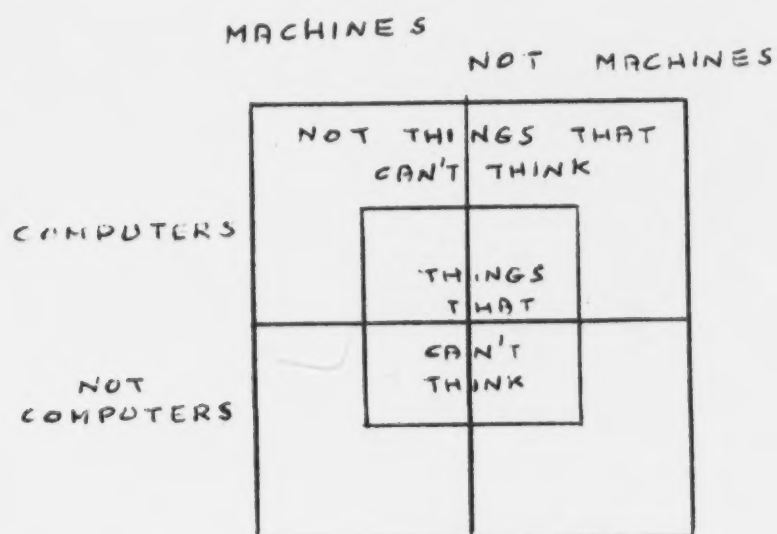
which was formalized as

- (1) All A are B
- (2) All B are C
- (3) $\therefore$ All A are C

by setting A = the class of computers
 B = the class of machines
 C = the class of things that can't think.

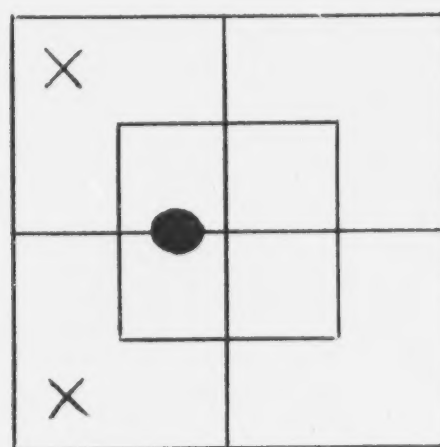
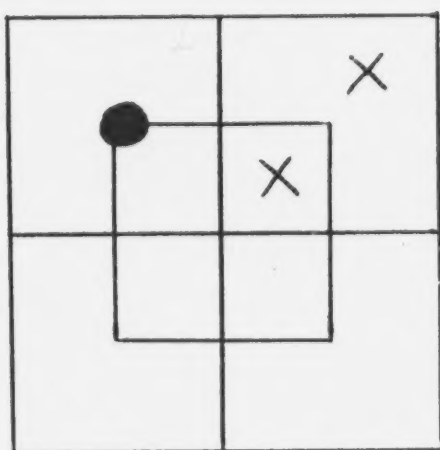
Note that we have made explicit the "all" which was implicit in the statements and rewritten the second as "All machines are things that can't think". This is the usual procedure in treating syllogisms formally. If the verb in a statement is different

from "are" or "is", we translate the predicate into the form "are things that ____", e.g., "Birds fly" becomes "Birds are things that fly". We treat all statements in syllogisms as being about classes rather than about actions. If we form the diagram



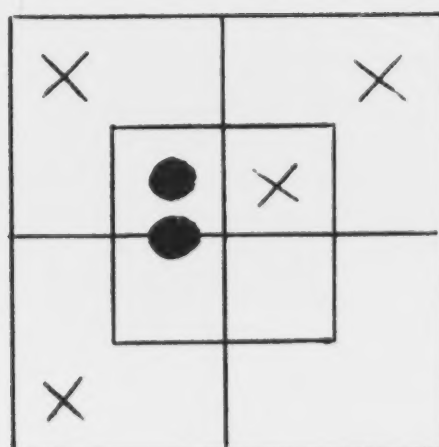
(1) becomes

and (2) is



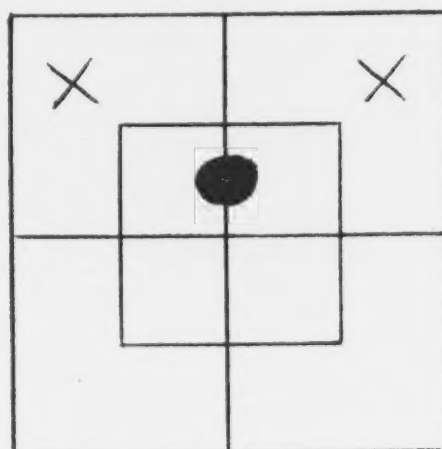
When these two diagrams are combined the x in the upper left corner forces the circle into the center square.

The combined diagram becomes



(The lower circle does not add any information to the diagram)

The conclusion of the argument is "All computers are things that can't think" which has the diagram



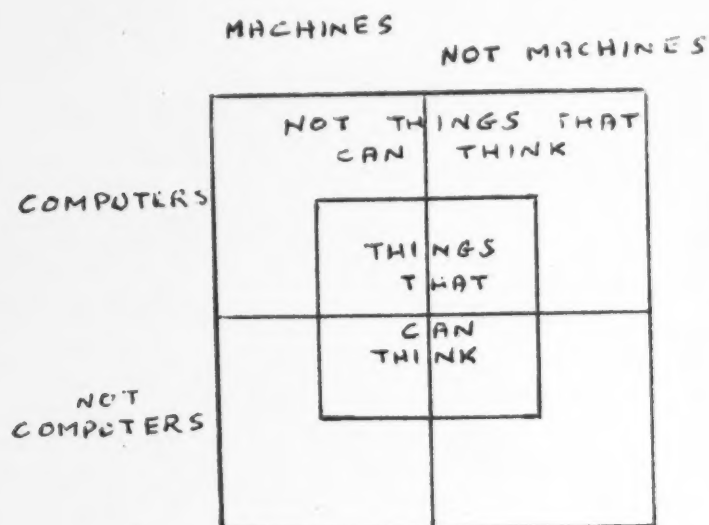
For this to be implied by the premises, the premises must give an x in the indicated places and a • in one of the two regions which intersect the circle in the diagram (or a circle intersecting both). As the combined diagram does satisfy these requirements, this syllogism is valid.

The same argument might be cast more naturally in a different syllogistic form if we make A = computers, B = machines, as before, but C = things that *can* think. Now statements 2 and 3 are replaced by (2') No machines are things that can think
 (3') No computers are things that can think.

The form of the argument now becomes

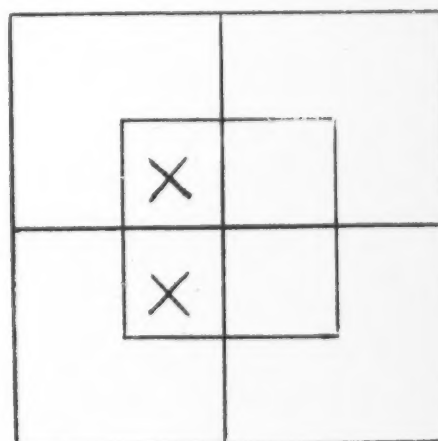
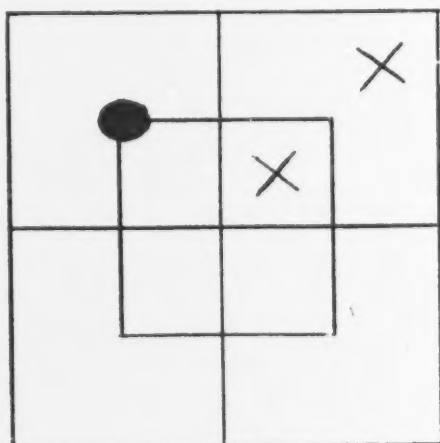
All A are B
 No B are C
 ∴ No A are C

If we now construct the diagram as follows:

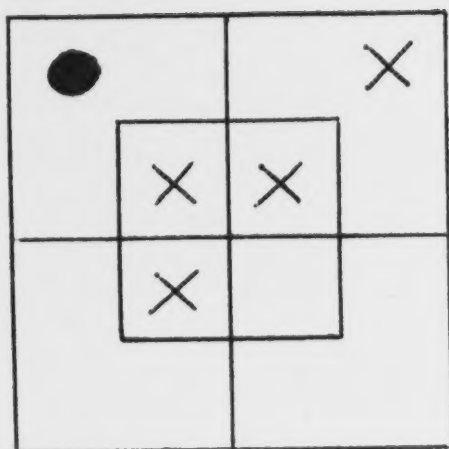


(1) is represented by

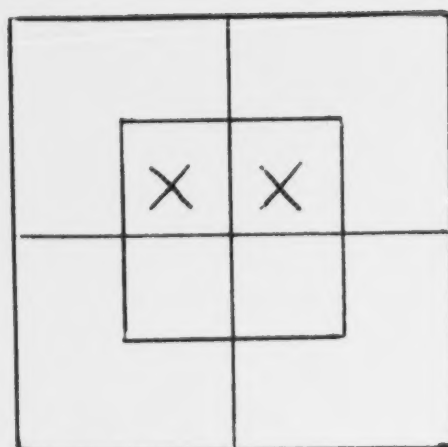
and (2') by



The combined diagram is



The diagram of (3') is



The syllogism is valid since the combined diagram does contain x's in the two indicated places.

Our second syllogism was

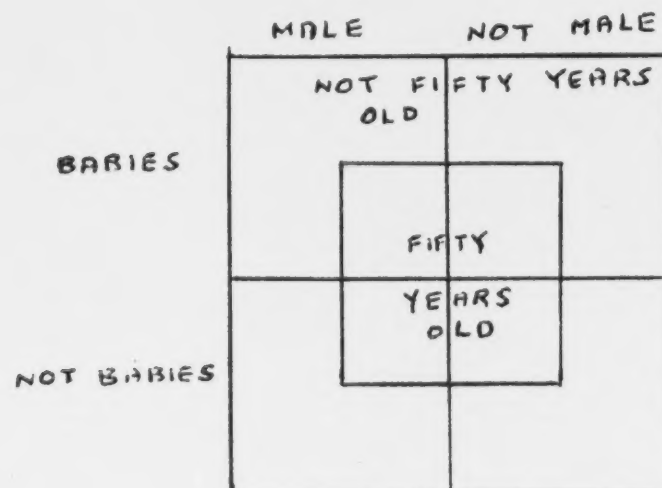
- (4) Some babies are male
- (5) Some males are fifty years old
- (6) $\therefore$ Some babies are fifty years old

This has the form

Some A are B
Some B are C
 $\therefore$ Some A are C

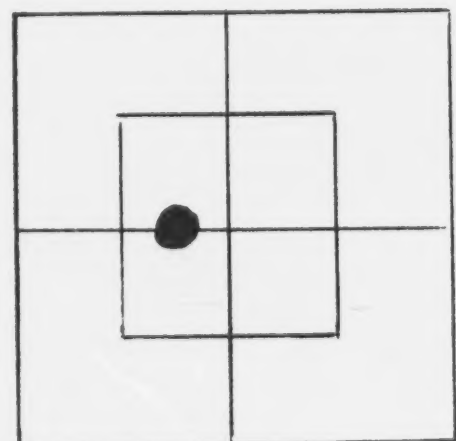
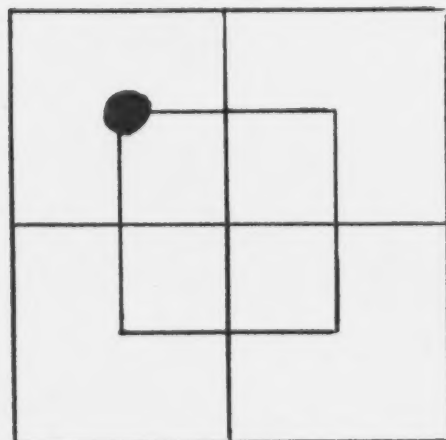
where A is the class of babies, B the class of males, and C the class of fifty-year-old people.

The diagram is

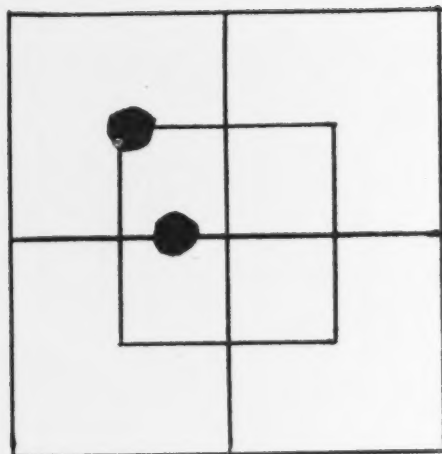


(4) is represented by

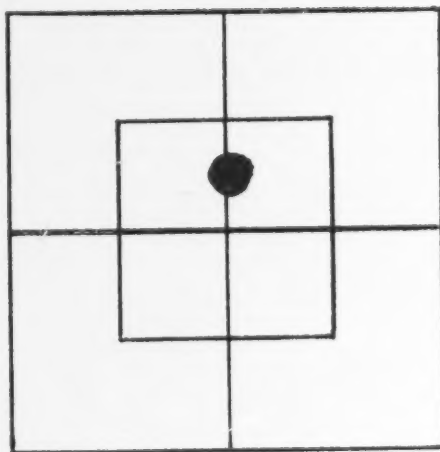
and (5) by



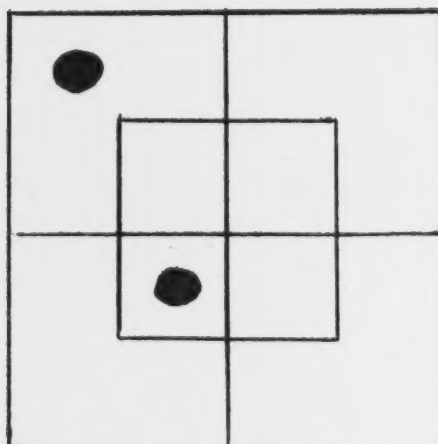
The combined diagram is



The diagram for (6) is



For the argument to be valid, the combined diagram must contain a circle in one of the two areas which intersect the circle in (6) or a circle intersecting both. This is not the case. In fact, the combined diagram is consistent with



which does not satisfy the conclusion.

Some inferences which contain statements about particular objects rather than classes can be treated as categorical syllogisms.

For example,

John is a boy
All boys like ice cream
 $\therefore$ John likes ice cream

can be formalized as

All A are B
All B are C
 $\therefore$ All A are C

if we take A to be the class consisting of the single object "John", B the class of boys, and C the class of people who like ice cream.

An argument with a particular premise which is invalid is

John likes ice cream
Some people who like ice cream are fat
 $\therefore$ John is fat

taking A = "John", as before,
 B = people who like ice cream,
 C = fat people.

This argument can be put in the form

$$\begin{array}{l} \text{All A are B} \\ \text{Some B are C} \\ \hline \therefore \text{All A are C} \end{array}$$

(Here we are using our knowledge that "John" usually refers to a person rather than a dog or some other thing.)

This syllogism can be seen to be invalid by constructing the Lewis Carroll diagrams for these three statements.

A syllogism is valid or invalid because of its form, not because of the truth or falsity of its premises and conclusion. For instance, the syllogism

$$\begin{array}{l} \text{All cows are things that fly} \\ \text{All things that fly are birds} \\ \hline \therefore \text{All cows are birds} \end{array}$$

is of the form

$$\begin{array}{l} \text{All A are B} \\ \text{All B are C} \\ \hline \text{All A are C} \end{array}$$

and hence valid, although its premises and conclusion are all false.

On the other hand,

$$\begin{array}{l} \text{Some people are men} \\ \text{Some men have beards} \\ \hline \therefore \text{Some people have beards} \end{array}$$

is of the form

$$\begin{array}{l} \text{Some A are B} \\ \text{Some B are C} \\ \hline \therefore \text{Some A are C} \end{array}$$

which is an invalid syllogism, even though its premises and conclusion are all true.

A syllogism of a given form is invalid if it is possible to produce another syllogism of the same form with true premises and a false conclusion. One can prove a syllogism invalid by generating counter-examples in this way.

Syllogistic arguments play a very important role in reasoning but are usually not stated explicitly in the forms discussed above.

2. Generalization of Lewis Carroll Diagrams

It is possible to extend Lewis Carroll diagrams to relations among four or more classes, so that we might treat extended syllogisms such as:

Some people are babies.
All babies are young animals.
All young animals are delicate.
∴ Some people are delicate.

However, when the diagrams are thus extended, they lose their transparency. The feature which makes them especially useful for the three classes of ordinary syllogisms is the fact that an O "sitting on the fence" actually overlaps contiguous areas. This is not always possible for diagrams representing a larger number of classes.

For a generalized syllogism, which may contain a large number of classes and premises, we will replace the Lewis Carroll diagram by a linear array, simply a list of all possible combinations of classes and class complements. For example, the list for two classes, A B, is:

A B
A not-B
not-A B
not-A not-B

The list corresponding to three classes has eight entries and the list for four classes has sixteen. The markers which were used in the Lewis Carroll diagrams are now placed next to the corresponding entries in the list. Thus, we represent the statement "no A are B" by placing an X next to each entry in the list containing A and B unnegated.

e.g., A B C X
 A B not-C X

The statement "some A are B" was previously represented by a circle on the fence between the regions A, B, C and A, B, not-C. Now the regions have been replaced by entries on a list with no topological significance. Further, there may be more than three classes, so that the statement above might refer to several entries, e.g., A B C D, A B C not-D, A B not-C D, and A B not-C not-D. We need a new nongraphical notation to represent this situation. We can put a marker, say 1, next to each entry representing a class with A, B unnegated. In the above example, each of the four entries given would receive a "1". If another statement, say "some B are not-C", put another 0 "on the fence", we might use a "2" to indicate each class intersected by this new 0.

The increase in number of classes also forces some operational changes. In the Lewis Carroll diagrams an 0 on the fence always intersected exactly two regions. If a later statement put an X in one of the regions, the 0 was automatically taken off the fence and placed in the other region, since there was no other possibility. This is no longer the case when a "fence" joins more than two regions. In the above example, if an X were placed next to A B C D (say by the statement "no C are D"), the 1 next to this entry would be eliminated, but three 1's would still remain.

Thus, the 1 is still on the fence. It is not replaced by an 0 unless a single 1 remains on the list.

3. Generating Logic "Diagrams"

We are ready to translate the processes involved in the use of generalized Lewis Carroll diagrams into the form of computer programs. There are four major parts to this undertaking. First, in this section, we will discuss the creation of generalized diagrams, given a list of objects. Then we can write procedures which "mark" diagrams to represent the statements given as their input. Once this is done, and a diagram representing the set of all premisses is made, we can then test the validity of a further statement with respect to these premisses. These three preceding steps, in fact, create a complete, consistent structure. Such a structure, however, will only accept statements and objects in a rigidly standardized form. Our fourth and last step will be, therefore, to write a set of procedures which enable us to deal with a more "relaxed" set of statements and objects.

We have seen how the Lewis Carroll diagrams can be generalized, by simply making a list of the names of the regions, without placing any topological interpretations on such a list. The "contents" of each region can then be denoted simply by associating these with the appropriate name. This is done most simply by making the contents the THING of the region name. The list of region names is best written as a LOGO sentence with commas (or some

other demarcation) between the names. In that way, we can still use sentences for region names. This convention does not permit the individual objects within a region name (NOT GREEN CATS, for example) to be sentences, but, rather than use another symbol as a further level of demarcation, we will make these objects into LOGO words by putting dashes in place of the spaces within them. (NOT-GREEN-CATS, for example). Thus, if we are given a set of objects we first insert dashes in each:

```
TO DASH /OBJECT/
10 TEST WORDP /OBJECT/
20 IF TRUE OUTPUT /OBJECT/
30 OUTPUT DASHSENTENCE /OBJECT/
END
```

```
TO DASHSENTENCE /SENTENCE/
10 TEST IS COUNT /SENTENCE/ 1
20 IF TRUE OUTPUT FIRST /SENTENCE/
```

(If there is just one word in /SENTENCE/, we get this remaining word using FIRST)

```
30 OUTPUT WORD WORD
    FIRST /SENTENCE/
    "-"
    DASHSENTENCE BUTFIRST /SENTENCE/
END
```

```
+PRINT DASH "IS A MAN"
IS-A-MAN
+PRINT DASH "ARISTOTLE"
ARISTOTLE
+
```

The really important task, at this early point, is construction of the region name list. We will have a list of objects, each of which is a LOGO word (we ensure this by using DASH). The procedure REGIONLIST takes /OBJECTS/ as input and outputs the list of all possible combinations, each containing every object

or its negation (NOT-object). These possibilities are to be separated by commas. Let us consider an example where the objects are taken as A, B, C and we place each possibility on a separate line.

A	B	C
A	B	NOT-C
A	NOT-B	C
A	NOT-B	NOT-C
NOT-A	B	C
NOT-A	B	NOT-C
NOT-A	NOT-B	C
NOT-A	NOT-B	NOT-C

There are a number of ways to systematically generate this set of combinations. Furthermore, the result of each method can be looked at in different ways, depending on the patterns one sees in the result. The scheme used in the example above was to find all possibilities, changing the rightmost character (lines 1, 2), to repeat these possibilities, having changed the next character (lines 3 and 4), and so on. This process is very reminiscent of counting, if one thinks of an automobile odometer, for example, and there is, in fact, an equivalence between the scheme used and counting in binary mode. Counting to base 2 is governed by the rules $0 + 0 = 0$ $0 + 1 = 1 + 0 = 1$ and $1 + 1 = 10$ where "carrying" is exactly as in base ten. Thus, putting zeros in front of the binary numbers (to give 3 digits) as we count from 0 to 7,

```

000
001
010
011
100
101
110
111

```

The zeros and ones exactly reproduce the pattern of "NOT"s and absence of NOTs if we make the correspondence $0 \leftrightarrow$ absence of NOT $1 \leftrightarrow$ NOT.

Thus, counting in binary (which is very easy to program) gives us the pattern we need for our combination generating problem.

To get the region name list for /N/ objects we need only count from $\underbrace{00\dots0}_{/N/}$ to $\underbrace{1111\dots1}_N$ in binary, then use each number to create a region name, using the correspondence given above. Thus, we write a procedure, which, given a binary number, outputs the next binary number.

```

TO BINARYNEXT /NUMBER/
1Ø TEST EMPTY /NUMBER/
2Ø IF TRUE OUTPUT 1
3Ø TEST ZEROP LAST /NUMBER/      (If the last digit is zero, just
4Ø IF TRUE OUTPUT WORD           change it to 1)
    BUTLAST /NUMBER/
    1
5Ø OUTPUT WORD                    (If the last digit is 1 (the
    BINARYNEXT BUTLAST /NUMBER/ only other possibility) change
    Ø                             it to zero and add 1 to the
                                binary number, BUTLAST
                                /NUMBER/.)
END

```

This process continues until either a zero is found in which case it is replaced by a 1, or until the last digit of the original number has been passed, in which case a 1 is placed in front.

```

←PRINT BINARYNEXT ØØ
Ø1
←PRINT BINARYNEXT Ø1
1Ø
←PRINT BINARYNEXT 11
1ØØ
←

```

We want to start the process with /N/ zeros so,

```

TO ZERO /N/
1Ø TEST IS /N/ Ø
2Ø IF TRUE OUTPUT /EMPTY/
---3Ø OUTPUT WORD OF
      Ø
      ZERO (DIFF /N/ AND 1)
END

```

```

←PRINT ZERO "3"
ØØØ
←PRINT BINARYNEXT BINARYNEXT ZERO 3
Ø1Ø

```

We use these two procedures in two further procedures, one of which, REGIONLIST, starts things off, and the other, ADDTOLIST, adds one combination at a time to the list of region names.

```

TO ADDTOLIST /OBJECTS/ /BINARY #/
1Ø TEST GREATERP (COUNT /BINARY #/)
      (COUNT /OBJECTS/)      (If the number of digits in the
2Ø IF TRUE OUTPUT /EMPTY/      binary number exceeds the number
                                of objects, we are finished)
3Ø OUTPUT SENTENCE SENTENCE
      NOTT /BINARY #/ /OBJECTS/ (NOTT gives the combination of
      ", "                      /OBJECTS/ corresponding to /BINARY #/)
      ADDTOLIST /OBJECTS/
      (BINARYNEXT /BINARY#/)
END

```

The procedure NOTT /NUMBER/ /LIST/, which applies the correspondence between NOTs and a binary number to /LIST/ is straightforward: If the first digit of /NUMBER/ is Ø, we leave the first element of list unchanged. Otherwise, if it is 1, we put a "NOT-" in front of it. We repeat the process until the list is exhausted.

```

TO NOTT /NUMBER/ /LIST/
1Ø TEST IS /LIST/ /EMPTY/           (If the list is /EMPTY/, we are
2Ø IF TRUE OUTPUT /EMPTY/           finished)
3Ø TEST IS FIRST /NUMBER/ Ø
4Ø IF TRUE OUTPUT SENTENCE
    FIRST /LIST/
    NOTT (BUTFIRST /NUMBER/) (BUTFIRST /LIST/)
5Ø OUTPUT SENTENCE
    WORD "NOT-" (FIRST /LIST/)
    NOTT (BUTFIRST /NUMBER/) (BUTFIRST /LIST/)
END

```

```

←PRINT NOTT "Ø11" "DOGS CATS ELEPHANTS"
DOGS NOT-CATS NOT-ELEPHANTS

```

Now, REGIONLIST is easy:

```

TO REGIONLIST /OBJECTS/
1Ø OUTPUT ADDTOLIST /OBJECTS/ (ZERO OF COUNT OF /OBJECTS/)
END

```

```

←PRINT REGIONLIST "A B C"
A B C , A B NOT-C , A NOT-B C , A NOT-B NOT-C , NOT-A B C , NOT-A B
NOT-C , NOT-A NOT-B C , NOT-A NOT-B NOT-C ,

```

```

←PRINT REGIONLIST "CATS DOGS MONKEYS ELEPHANTS CAMELS"
CATS DOGS MONKEYS ELEPHANTS CAMELS , CATS DOGS MONKEYS ELEPHANTS
NOT-CAMELS , CATS DOGS MONKEYS NOT-ELEPHANTS CAMELS , CATS DOGS MONKEYS
NOT-ELEPHANTS NOT-CAMELS , CATS DOGS NOT-MONKEYS ELEPHANTS CAMELS , CATS
DOGS NOT-MONKEYS ELEPHANTS NOT-CAMELS , CATS DOGS NOT-MONKEYS
NOT-ELEPHANTS CAMELS , CATS DOGS NOT-MONKEYS NOT-ELEPHANTS NOT-CAMELS ,
CATS NOT-DOGS MONKEYS ELEPHANTS CAMELS , CATS NOT-DOGS MONKEYS ELEPHANTS
NOT-CAMELS , CATS NOT-DOGS MONKEYS NOT-ELEPHANTS CAMELS , CATS NOT-DOGS
MONKEYS NOT-ELEPHANTS NOT-CAMELS , CATS NOT-DOGS NOT-MONKEYS ELEPHANTS
CAMELS , CATS NOT-DOGS NOT-MONKEYS ELEPHANTS NOT-CAMELS , CATS NOT-DOGS
NOT-MONKEYS NOT-ELEPHANTS CAMELS , CATS NOT-DOGSNOT-MONKEYS
NOT-ELEPHANTS NOT-CAMELS , NOT-CATS DOGS MONKEYS ELEPHANTS CAMELS ,
NOT-CATS DOGS MONKEYS ELEPHANTS NOT-CAMELS , NOT-CATS DOGS MONKEYS
NOT-ELEPHANTS CAMELS , NOT-CATS DOGS MONKEYS NOT-ELEPHANTS NOT-CAMELS ,
NOT-CATS DOGS NOT-MONKEYS ELEPHANTS CAMELS , NOT-CATS DOGS NOT-MONKEYS
ELEPHANTS NOT-CAMELS , NOT-CATS DOGS NOT-MONKEYS NOT-ELEPHANTS CAMELS ,
NOT-CATS DOGS NOT-MONKEYS NOT-ELEPHANTS NOT-CAMELS , NOT-CATS NOT-DOGS
MONKEYS ELEPHANTS CAMELS , NOT-CATS NOT-DOGS MONKEYS ELEPHANTS
NOT-CAMELS , NOT-CATS NOT-DOGS MONKEYS NOT-ELEPHANTS CAMELS , NOT-CATS
NOT-DOGS MONKEYS NOT-ELEPHANTS NOT-CAMELS , NOT-CATS NOT-DOGS
NOT-MONKEYS ELEPHANTS CAMELS , NOT-CATS NOT-DOGS NOT-MONKEYS ELEPHANTS
NOT-CAMELS , NOT-CATS NOT-DOGS NOT-MONKEYS NOT-ELEPHANTS CAMELS ,
NOT-CATS NOT-DOGS NOT-MONKEYS NOT-ELEPHANTS NOT-CAMELS ,

```

This completes the first stage of our program--we can easily create region name lists. These lists are not easy to work with directly in LOGO since the separators of interest are commas, and not spaces. FIRST, for example, will give us the first element of the first region name. We therefore write a few simple procedures which will be useful in working with our new kind of list.

```

TO PULL /LIST/                                     (gives everything to first comma)
10 TEST EITHER
    EMPTY /LIST/
    IS FIRST /LIST/ ","
20 IF TRUE OUTPUT /EMPTY/
30 OUTPUT SENTENCE
    FIRST /LIST/
    PULL BUTFIRST /LIST/
END

```

```

TO REST /LIST/                                     (gives everything after first comma)
10 TEST EITHER
    EMPTY /LIST/
    IS FIRST /LIST/ ","
20 IF TRUE OUTPUT BUTFIRST /LIST/
30 OUTPUT REST (BUTFIRST /LIST/)
END

```

```

TO NUMBER /LIST/                                   (counts "region names" by counting
10 TEST EMPTY /LIST/                               commas -- this is correct since
20 IF TRUE OUTPUT 0                                 the list ends with a comma.)
30 TEST IS (FIRST /LIST/) ","
40 IF TRUE OUTPUT SUM
    1
    NUMBER (BUTFIRST /LIST/)
50 OUTPUT NUMBER (BUTFIRST /LIST/)
END

```

These procedures enable us, for example, to write a simple procedure NICEPRINT to print the list of regions "nicely" by putting each on a separate row. To make NICEPRINT even more useful, next to each

region we print its "contents", given by the THING of the region name. We have not yet "marked" any of the regions, so at the moment, the contents are all /EMPTY/.

```
TO NICEPRINT /LIST/
1Ø TEST EMPTY REST /LIST/
2Ø IF TRUE STOP
3Ø TYPE PULL /LIST/
4Ø TYPE ", "
5Ø PRINT THING (PULL /LIST/)
6Ø NICEPRINT (REST /LIST/)
END
```

(TYPE prints but does not go to the beginning of the next line)

```
*NICEPRINT REGIONLIST "FRED HARRY JANE SALLY"
FRED HARRY JANE SALLY,
FRED HARRY JANE NOT-SALLY,
FRED HARRY NOT-JANE SALLY,
FRED HARRY NOT-JANE NOT-SALLY,
FRED NOT-HARRY JANE SALLY,
FRED NOT-HARRY JANE NOT-SALLY,
FRED NOT-HARRY NOT-JANE SALLY,
FRED NOT-HARRY NOT-JANE NOT-SALLY,
NOT-FRED HARRY JANE SALLY,
NOT-FRED HARRY JANE NOT-SALLY,
NOT-FRED HARRY NOT-JANE SALLY,
NOT-FRED HARRY NOT-JANE NOT-SALLY,
NOT-FRED NOT-HARRY JANE SALLY,
NOT-FRED NOT-HARRY JANE NOT-SALLY,
NOT-FRED NOT-HARRY NOT-JANE SALLY,
NOT-FRED NOT-HARRY NOT-JANE NOT-SALLY,
```

4. Marking the "Diagrams"

There are two different operations required to represent a statement, or sequence of statements on a diagram, whether it is of the Lewis Carroll type or a more general list one. We must be able to "mark off" regions, by placing X's in them. We do this by making the THING of the region name "X". We also must be able to denote occupancy or possible occupancy of a region. If there is only one available region satisfying the given constraints in the statement(s), an "0" is placed within it. If several regions are possible, they are "joined" by placing the same number in the THING of each. Any conflict between 0s and Xs means that a contradiction has been found.

Let us first consider the algorithm required for "joining" or, in terms of Lewis Carroll diagrams, placing an "0" on the fence. We take each region name and determine whether it contains the combination of objects and their negation as in the statement we are joining. For example, given the three objects A, B, C and the list of 8 names that they generate, we determine that the elements A NOT-B are contained within the two regions A NOT-B C, A NOT-B NOT-C. We make a list of all such "valid" names, which do not contain an "X". If there is exactly one such region, we *occupy* it by adding an "0" to its THING. If there is more than one region possible, we *include* the current join number within the THING of each. Of course, if no regions are possible, a contradictory set of premisses is involved and we can go no further.

Our first step is to write a procedure SUBSETP, which, given two sets /A/ and /B/ as inputs, outputs TRUE or FALSE as all elements of /A/ are or are not contained among those of /B/. SUBSETP in

turn depends on CP /EL/ /LIST/ which determines whether a given single element /EL/ is contained in a given list, /LIST/.

```

TO SUBSETP /A/ /B/
10 TEST EMPTYP /A/
20 IF TRUE OUTPUT "TRUE"           (is the first element of /A/ contained
30 TEST CP (FIRST /A/) /B/         in /B/?)
40 IF FALSE OUTPUT "FALSE"         (if not /A/ is not a subset of /B/)
50 OUTPUT SUBSETP (BUTFIRST /A/) /B/ (if yes, continue the process
                                     with the rest of /A/)
END

```

```

TO CP /ELEMENT/ /LIST/
10 TEST EMPTYP /LIST/              (have we gone through the whole list?)
20 IF TRUE OUTPUT "FALSE"          (if so /ELEMENT/ is not contained)
30 TEST IS /ELEMENT/
   FIRST /LIST/
40 IF TRUE OUTPUT "TRUE"
50 OUTPUT CP /ELEMENT/ (BUTFIRST /LIST/)
END

```

```

←PRINT SUBSETP "A B C D" "D C B A"
TRUE
←PRINT SUBSETP "PLEAT" "ELEPHANT"
TRUE

```

We next write the main procedure JOIN /ELEMENTS/ /JOIN #/ /LIST/, and its sub-procedures. JOIN is virtually a line by line translation of the joining algorithm just stated.

```

TO JOIN /ELEMENTS/ /JOIN #/ /LIST/
10 MAKE "COMMON REGIONS" COMMON OF /ELEMENTS/ /LIST/
   (make a list of regions on /LIST/ without "X"s, having
   /ELEMENTS/ as subset)
20 TEST IS NUMBER /COMMON REGIONS/ 0
30 IF TRUE EXIT "CONTRADICTION" (EXIT is a LOGO built-in which
   halts execution and causes its
   input to be printed)
40 TEST IS NUMBER /COMMON REGIONS/ 1
50 IF TRUE OCCUPY /JOIN #/ (PULL /LIST/)
   (if there is exactly one region,
   place an "0" within it)
60 IF TRUE STOP
70 INCLUDE /JOIN #/ /COMMON REGIONS/ (otherwise place /JOIN #/ within
   each of the common regions.)
END

```

The procedures remaining to be written are COMMON, OCCUPY and INCLUDE, all mentioned in our description of the joining algorithm.

```

TO COMMON /ELEMENTS/ /LIST/
10 TEST EMPTY /LIST/
20 IF TRUE OUTPUT /EMPTY/
30 TEST BOTH
    SUBSETP /ELEMENTS/ (PULL /LIST/) (PULL /LIST/ is the first
    NOT CP "X" (THING OF PULL /LIST/) region name on /LIST/. If
40 IF TRUE OUTPUT SENTENCE SENTENCE /ELEMENTS/ is a subset of
    PULL /LIST/ the first region name and
    "," the region does not contain
    COMMON /ELEMENTS/ (REST /LIST/) "X", add the first region
50 OUTPUT COMMON /EL/ (REST /LIST/) to the output list)
END (Otherwise, just repeat
rest of /LIST/)

```

NOT, on line 30 simply outputs the negation of its input,

```

TO NOT /INPUT/
10 TEST IS INPUT "TRUE"
20 IF TRUE OUTPUT "FALSE"
30 OUTPUT "TRUE"
END

```

The following example illustrates the use of COMMON.

```

-MAKE "REGIONS" REGIONLIST "CATS HATS BATS"
-NICEPRINT /REGIONS/
CATS HATS BATS,
CATS HATS NOT-BATS,
CATS NOT-HATS BATS,
CATS NOT-HATS NOT-BATS,
NOT-CATS HATS BATS,
NOT-CATS HATS NOT-BATS,
NOT-CATS NOT-HATS BATS,
NOT-CATS NOT-HATS NOT-BATS,
-PRINT COMMON "CATS HATS" /REGIONS/
CATS HATS BATS , CATS HATS NOT-BATS ,
-PRINT COMMON "NOT-BATS" /REGIONS/
CATS HATS NOT-BATS , CATS NOT-HATS NOT-BATS , NOT-CATS HATS NOT-BATS ,
NOT-CATS NOT-HATS NOT-BATS ,
-

```

We look ahead and write a version of OCCUPY which is just a little more powerful than we require at the moment. Right now, we need only have OCCUPY place an 0 in the THING of the region name given as input. Later, however, we will also want to change a "join" to an "0" when marking "X"s reduces the number of possible regions to 1. In this latter case we want to DELETE an inputted "join number" as well as write in an "0".

```

TO OCCUPY /JOIN #/ /REGION/
10 MAKE /REGION/
      SENTENCE
      "0"
      DELETE /JOIN #/ (THING OF /REGION/)
END

```

DELETE goes through /LIST/, deleting each occurrence of /ELEMENT/.

```

TO DELETE /ELEMENT/ /LIST/
10 TEST EMPTY /LIST/
20 IF TRUE OUTPUT /EMPTY/
30 TEST IS /ELEMENT/ (FIRST /LIST/)
40 IF TRUE OUTPUT DELETE (BUTFIRST /LIST/)
50 OUTPUT SENTENCE
      FIRST /LIST/
      DELETE (BUTFIRST /LIST/)
END

```

```

←DELETE "B" "A B C B"
A C

```

To complete the joining algorithm we need only write INCLUDE /NUMBER/ /LIST/. INCLUDE simply includes /NUMBER/ as a member of the THING of each region name on /LIST/.

```

TO INCLUDE /NUMBER/ /LIST/
1Ø TEST EMPTY /LIST/
2Ø IF TRUE STOP
3Ø MAKE PULL /LIST/
    SENTENCE
        THING OF PULL /LIST/
        /NUMBER/
4Ø INCLUDE /NUMBER/ (REST /LIST/)
END

```

And we have completed our implementation of the joining algorithm. We try it out; using the list /REGIONS/ we generated just before:

```

- JOIN "CATS HATS" 1 /REGIONS/
- JOIN "CATS NOT-BATS" 2 /REGIONS/
- NICEPRINT /REGIONS/
CATS HATS BATS,1
CATS HATS NOT-BATS,1 2
CATS NOT-HATS BATS,
CATS NOT-HATS NOT-BATS,2
NOT-CATS HATS BATS,
NOT-CATS HATS NOT-BATS,
NOT-CATS NOT-HATS BATS,
NOT-CATS NOT-HATS NOT-BATS,
-

```

The other operation we will have to perform on our extended syllogism diagrams is that of "marking off" regions to indicate the impossibility of their being occupied. As in the case of joining, we will generally have several regions to be marked off, as the result of a given premiss, each containing the elements specified in the premiss. If a region does not contain anything, we need merely put an "X" in it. If the region contains "0", we have a contradiction. The only difficult case is when the region contains one or more join numbers. When this is true, we use a procedure UNJOIN to "remove" all joins from this region. To remove each join number, /JOIN #/, we make a list, using JOINLIST, of all other regions containing /JOIN #/. If there is exactly one region on this list, /JOIN #/ there is replaced by "0". If

there is more than one region on the list, these other regions are not affected. In either case /JOIN #/ can then be removed from the given region. When all join numbers have been removed from the given region, an "X" can be inserted.

For example, using the small region list corresponding to "A B", marking off all regions containing the element "A" in

A	B	,	1
A	NOT-B	,	1
NOT-A	B	,	1
NOT-A	NOT-B	,	1

results in

A	B	,	X
A	NOT-B	,	X
NOT-A	B	,	1
NOT-A	NOT-B	,	1

and, doing the same in

A	B	,	1
A	NOT-B	,	1
NOT-A	B	,	1
NOT-A	NOT-B	,	

results in

A	B	,	X
A	NOT-B	,	X
NOT-A	B	,	0
NOT-A	NOT-B	,	

Thus, to implement our "marking off" algorithm we must write these procedures UNJOIN and JOINLIST, to be used as part of the main procedure which we call MARKOFF. All other subprocedures have already been written in the process of implementing joining:

```

TO MARKOFF /OBJECTS/ /LIST/ /REGION LIST/
10 TEST EMPTY /LIST/
20 IF TRUE STOP
30 TEST SUBSETP /OBJECTS/ (PULL /LIST/)
40 IF TRUE UNJOIN (PULL /LIST/) /REGION LIST/
50 MARKOFF /OBJECTS/ (REST /LIST/) /REGION LIST/
END

```

It is necessary, in the above to have two lists of region names. The first, /LIST/, is the one that MARKOFF goes through, using PULL and REST, looking for regions to UNJOIN. We also need at all times, a complete list of all regions, /REGION LIST/ so that UNJOIN will take all regions into account.

```

TO UNJOIN /REGION/ /REGION LIST/
10 TEST IS FIRST OF (THING /REGION/) "0" (if /REGION/ contains "0"
20 IF TRUE EXIT "CONTRADICTION"          as first element, we have
                                          a contradiction)
30 TEST EMPTY (THING /REGION/)          (if /REGION/ is
                                          /EMPTY/, we place an "X"
                                          in it and are done)

40 IF TRUE MAKE /REGION/ "X"
50 IF TRUE STOP
60 MAKE "JOINS" JOINLIST FIRST (THING /REGION/) (otherwise we make
                                          /REGION LIST/ a list of all regions
                                          /REGION/ other than the given
                                          one, /REGION/, which
                                          contains the first
                                          number within /REGION/)

70 TEST IS NUMBER /JOINS/ 1              (if there is just one other
                                          region this join number is
                                          changed to "0")

80 IF TRUE OCCUPY
      FIRST (THING /REGION/)
      PULL /JOINS/
90 MAKE /REGION/ BUTFIRST (THING OF /REGION/) (we repeat, having
                                          eliminated the first
                                          element of the contents
                                          of /REGION/)

100 UNJOIN /REGION/ /REGION LIST/
END

```

```

TO JOINLIST /JOIN #/ /LIST/ /REGION/
10 TEST EMPTY /LIST/ (if we have gone through the entire
                        list of regions, we are finished)
20 IF TRUE OUTPUT /EMPTY/
30 TEST BOTH
    CP /JOIN #/ (PULL /LIST/)
    NOT IS (PULL /LIST/) /REGION/ (does the first region on
                                    /LIST/ contain /JOIN #/
                                    and is it other than /REGION/?)
40 IF TRUE OUTPUT SENTENCE SENTENCE
    PULL /LIST/
    " "
    JOINLIST /JOIN #/ (REST /LIST/) /REGION/
50 OUTPUT JOINLIST /JOIN #/ (REST /LIST/) /REGION/ (otherwise just
                                                    keep on going)
END

```

For example:

```

+MAKE "REGIONS" REGIONLIST "CATS DOGS"
+NICEPRINT /REGIONS/
CATS DOGS,
CATS NOT-DOGS,
NOT-CATS DOGS,
NOT-CATS NOT-DOGS,
+MARKOFF "CATS" /REGIONS/ /REGIONS/
+NICEPRINT /REGIONS/
CATS DOGS,X
CATS NOT-DOGS,X
NOT-CATS DOGS,
NOT-CATS NOT-DOGS,
+JOIN "DOGS" 1 /REGIONS/
+NICEPRINT /REGIONS/
CATS DOGS,X
CATS NOT-DOGS,X
NOT-CATS DOGS,0
NOT-CATS NOT-DOGS,
+MARKOFF "DOGS" /REGIONS/ /REGIONS/
CONTRADICTION
I WAS AT LINE 20 IN UNJOIN
+JOIN "CATS DOGS" 2 /REGIONS/
CONTRADICTION
I WAS AT LINE 30 IN JOIN
+

```

5. Diagrams and Premisses

We are now able to mark diagrams with considerable facility, using the two basic operations MARKOFF and JOIN. A set of premisses, however, is given not in terms of these primitive operations, but as statements of the four forms discussed earlier: ALL ___ ARE ___, SOME ___ ARE ___, NO ___ ARE ___, and NOT ALL ___ ARE ___. (The blanks in each of these statement forms are the names of the classes from which the region names are built. The use of DASH can be used to ensure that these class names are LOGO words.)

The relation between the four statement forms and our two primitive operations is easily given, using the letters A and B in place of the blanks used above:

ALL A ARE B	join A, B and markoff A, not-B
SOME A ARE B	join A, B
NO A ARE B	markoff A, B
NOT ALL A ARE B	join A, not-B

We can immediately see two useful consequences of initially limiting ourselves to these four standard forms: first, the beginning word of any statement of the above forms uniquely specifies the form, and second, given the form, the two class names are easily separated out, even if each consists of several LOGO words (before DASHing). Specifically, the second class, B, in each case is everything to the right of the word "ARE". Taking everything to the left of "ARE" and deleting the first word gives us the other class, A, in three of the four cases; for the other one, it is only necessary to eliminate the word "ALL" as well. A top-level procedure will, therefore, need subprocedures which take everything to the left and right of a word, /WORD/, in an expression, /EXP/.

```

TO LEFTS /WORD/ /EXP/
10 TEST EITHER IS (FIRST /EXP/) /WORD/
                IS /EXP/ /EMPTY/
20 IF TRUE OUTPUT /EMPTY/
30 OUTPUT SENTENCE
    FIRST /EXP/
    LEFTS /WORD/ BUTFIRST /EXP/
END

```

```

TC RIGHTS /WORD/ /EXP/
10 TEST EITHER
    IS /EXP/ /EMPTY/
    IS /WORD/ (FIRST /EXP/)
20 IF TRUE OUTPUT BUTFIRST /EXP/
30 OUTPUT RIGHTS /WORD/ (BUTFIRST /EXP/)
END

```

```

←PRINT LEFTS "G" (RIGHTS "B" "ABCDEFGH")
C D E F

```

These two procedures will enable us to sort out the two class names from a premiss. How do we then apply the proper combination of joins and markoffs which depend on the statement type? One way is to write four procedures, each having as name the first word of one of the four statement forms. The LOGO command DO can then be used to execute the first word of a given statement. We cannot do precisely this, since ALL is a reserved procedure name and we have already used NOT. To get around this, we simply add an "S", both to the procedure names and to the input of DO.

The class names /A/ and /B/ are given by a higher level procedure, ENTER, and thus need not be used as input names for our four procedures. This makes possible a simple DO command line (line 70). Thus, assuming a list of statements, separated by commas:

```

TO ENTER /STATEMENTS/ /REGION LIST/
1Ø TEST EMPTY /STATEMENTS/
2Ø IF TRUE STOP
3Ø MAKE "ALL REGIONS" /REGION LIST/
4Ø MAKE "SEN" PULL /STATEMENTS/
5Ø MAKE "A" DASH BUTFIRST (LEFTS "ARE" /SEN/)
6Ø MAKE "B" DASH BUTFIRST (RIGHTS "ARE" /SEN/)
7Ø DO WORD (FIRST /SEN/) "S"
8Ø ENTER (REST /STATEMENTS/) /REGION LIST/
END

```

```

TO ALLS
1Ø MARKOFF SENTENCE /A/ (WORD "NOT-" /B/)
    /ALL REGIONS/ /ALL REGIONS/
2Ø MAKE "JOIN #" (SUM /JOIN #/ 1)
3Ø JOIN (SENTENCE /A/ /B/) /ALL REGIONS/
END

```

```

TO NOTS
1Ø MAKE "A" BF BF BF BF /A/
2Ø MAKE "JOIN #" (SUM /JOIN #/ 1)
3Ø JOIN SENTENCE /A/ (WORD "NOT-" /B/)
    /JOIN #/ /ALL REGIONS/
END

```

(BF is the LOGO abbreviation for BUTFIRST)
 (We eliminate the ALL- which arises from the statement form)

```

TO SOMES
1Ø MAKE "JOIN #" (SUM /JOIN #/ 1)
2Ø JOIN (SENTENCE /A/ /B/)
    /JOIN #/ /ALL REGIONS/
END

```

```

TO NOS
1Ø MARKOFF (SENTENCE /A/ /B/) /ALL REGIONS/ /ALL REGIONS/
END

```

We are very nearly done. To simplify use of the procedures and to initialize /JOIN #/ at Ø and set up the region list, we write a top-level procedure, ACCEPT.

```

TO ACCEPT
1Ø PRINT "PREMISSES:"
2Ø MAKE "STATEMENTS" REQUEST
3Ø PRINT "CLASSES:"
4Ø MAKE "CLASSES" REQUEST
5Ø PRINT "NAME OF REGION LIST:"
6Ø MAKE "NAME" REQUEST
7Ø MAKE /NAME/ REGIONLIST /CLASSES/
8Ø MAKE "JOIN #" Ø
9Ø ENTER /STATEMENTS/ THING OF /NAME/
END

```

Note that we ask for a redundant set of information. The set of classes to be used can be deduced, without much difficulty from /STATEMENTS/. We will do this later, and much more, when we consider the general problems of accepting statements in a form more nearly approximating everyday discourse.

```

←ACCEPT
PREMISSES:
*ALL COMPUTERS ARE MACHINES , ALL MACHINES ARE THINGS THAT
CANNOT THINK ,
CLASSES:
*COMPUTERS MACHINES THINGS-THAT-CANNOT-THINK
NAME OF REGION LIST:
*AREAS
←NICEPRINT /AREAS/
COMPUTERS MACHINES THINGS-THAT-CANNOT-THINK,Ø 1
COMPUTERS MACHINES NOT-THINGS-THAT-CANNOT-THINK,X
COMPUTERS NOT-MACHINES THINGS-THAT-CANNOT-THINK,X
COMPUTERS NOT-MACHINES NOT-THINGS-THAT-CANNOT-THINK,X
NOT-COMPUTERS MACHINES THINGS-THAT-CANNOT-THINK,1
NOT-COMPUTERS MACHINES NOT-THINGS-THAT-CANNOT-THINK,X
NOT-COMPUTERS NOT-MACHINES THINGS-THAT-CANNOT-THINK,
NOT-COMPUTERS NOT-MACHINES NOT-THINGS-THAT-CANNOT-THINK,

```

This list can be compared with the Lewis Carroll diagram for the same premisses, given on page 11.

6. Testing Syllogisms

As the result of the procedures written in the previous section, we can now "mark" a diagram in a manner corresponding to a set of premisses. The final part of syllogism testing lies in determining whether the premisses, or equivalently, their diagrammatic representation justify a given conclusion. As before, we will find that once an algorithm for doing so is stated precisely in everyday language, the translation of the algorithm into LOGO procedures is not difficult.

The conclusion is a single statement and, therefore, can be decomposed into at most one markoff and one join. The marking off gives us a set of regions which cannot be occupied if the conclusion is to hold true. Therefore, all such regions must contain X's in the premiss diagram, if the syllogism is to be valid. Similarly, at least one of the set of regions "joined" by the conclusion must be occupied under the conditions of the premisses. This is certainly true if the premiss diagram has an O contained in any of these regions. It is also true if there is a set of regions "joined" by the premisses which is a proper subset of the "joined" regions in the conclusion. To illustrate this latter condition, if regions A B C D, A B C NOT-D, A B NOT-C D, A B NOT-C NOT-D are joined by the conclusion; and regions A B C D, and A B C NOT-D by the premisses; the required occupancy of one of the two regions of the premisses certainly implies occupancy of one of the four regions of the conclusion.

We will write two main procedures TESTJOINS and TESTMARKS, each to verify one of the above requirements. Then, as in Section 5, we will write a procedure TESTCONCLUSION which uses any one of four subprocedures TESTALL, TESTSOME, TESTNO, and TESTNOT to call a suitable version of TESTJOINS and/or TESTMARKS, depending on the form of the conclusion statement.

Let us first consider checking the markoffs, this being somewhat the easier. The simplest approach is to generate a list of all regions containing the classes of the conclusion. We can then generate the list of all marked regions of the premiss diagram. If the latter contains the former, TESTMARK outputs "TRUE", and otherwise, "FALSE". Thus, SUBLIST /ELEMENTS/ /REGION LIST/ outputs the list of all region names in /REGION LIST/ containing /ELEMENTS/.

```

TO SUBLIST /ELEMENTS/ /REGION LIST/
10 TEST EMPTY /REGION LIST/
20 IF TRUE OUTPUT /EMPTY/
30 TEST SUBSETP /ELEMENTS/ (PULL /LIST/)
40 IF TRUE OUTPUT SENTENCE SENTENCE
    PULL /LIST/
    " "
    SUBLIST /ELEMENTS/ (REST /REGION LIST/)
50 OUTPUT SUBLIST /ELEMENTS/ (REST /REGION LIST/)
END

```

This will enable us to find the regions that the conclusion requires to be marked off. Now, let us write COLLECTMARKS to get the list of all regions actually marked in the premiss diagram.

```

TO COLLECTMARKS /LIST/
10 TEST EMPTY /LIST/
20 IF TRUE OUTPUT /EMPTY/
30 TEST CP "X" THING (PULL /LIST/) (Does the first region on
                                   /LIST/ contain "X"?)
40 IF TRUE OUTPUT SENTENCE SENTENCE
    PULL /LIST/
    " "
    COLLECTMARKS (REST /LIST/)
50 OUTPUT COLLECTMARKS (REST /LIST/)
END

```

Now that we can generate the lists required, using SUBLIST and COLLECTMARKS, we must write a procedure to check whether one

list is contained in the other. We cannot simply use SUBSETP -- that procedure only checks two inputs to see if all the words in one LOGO sentence are contained within the other. Here, with region names, the important separators are the commas. A B , A NOT-B is quite different from A C , A NOT-C , B C , NOT-B C. Nevertheless SUBSETP would say the latter contains the former (which is true only in the sense of disjoint LOGO words). Thus, we write a new procedure ONP /ONE REGION/ /LIST/ which checks if /ONE REGION/, a single region name, is contained among /LIST/.

```
TO ONP /ONE REGION/ /LIST/
10 TEST EMPTY /LIST/
20 IF TRUE OUTPUT "FALSE"
30 TEST IS /ONE REGION/ (PULL /LIST/)
40 IF TRUE OUTPUT "TRUE"
50 OUTPUT ONP /ONE REGION/ (REST /LIST/)
END
```

We use ONP in WITHINP, which checks if several names -- /REGIONS/ -- are contained in /LIST/.

```
TO WITHINP /REGIONS/ /LIST/      (Goes through /REGIONS/ one by
                                one, using ONP)
10 TEST EMPTY /REGIONS/
20 IF TRUE OUTPUT "TRUE"
30 TEST ONP (PULL /REGIONS/) /LIST/
40 IF FALSE OUTPUT "FALSE"
50 OUTPUT WITHINP (REST /REGIONS/) /LIST/
END
```

```
<PRINT WITHINP "A B , A C" REGIONLIST "A B C"
TRUE
```

And now we have all the pieces for TESTMARKS:

```
TO TESTMARKS /ELEMENTS/ /REGION LIST/
10 OUTPUT WITHINP
   SUBLIST /ELEMENTS/ /REGION LIST/
   COLLECTMARKS /REGION LIST/
END
```

We try TESTMARKS with the set of premisses diagrammed on Page 39:

```
+NICEPRINT /AREAS/  
COMPUTERS MACHINES THINGS-THAT-CANNOT-THINK,Ø 1  
COMPUTERS MACHINES NOT-THINGS-THAT-CANNOT-THINK,X  
COMPUTERS NOT-MACHINES THINGS-THAT-CANNOT-THINK,X  
COMPUTERS NOT-MACHINES NOT-THINGS-THAT-CANNOT-THINK,X  
NOT-COMPUTERS MACHINES THINGS-THAT-CANNOT-THINK,1  
NOT-COMPUTERS MACHINES NOT-THINGS-THAT-CANNOT-THINK,X  
NOT-COMPUTERS NOT-MACHINES THINGS-THAT-CANNOT-THINK,  
NOT-COMPUTERS NOT-MACHINES NOT-THINGS-THAT-CANNOT-THINK,
```

```
+PRINT TESTMARKS "COMPUTERS NOT-MACHINES" /AREAS/  
TRUE  
+PRINT TESTMARKS "COMPUTERS" /AREAS/  
FALSE
```

Now we consider TESTJOINS /ELEMENTS/ /REGION LIST/. We are interested in the list of all regions whose names contain /ELEMENTS/, just as we were in the case of TESTMARKS and we can again use SUBLIST to find them. Our present criterion of validity is more complicated than in the previous case. We must find if any of the regions given by SUBLIST /ELEMENTS/ /REGION LIST/ contain an "O" -- if so we are finished. If not, we look at all join numbers contained in those regions. Our objective is to find one, which joins no regions outside this list, given by SUBLIST. If we cannot, the join given by the conclusion is not implied by the premisses.

We first get a list of everything contained in the regions on a given /LIST/.

```
TO COLLECT /LIST/  
1Ø TEST EMPTY /LIST/  
2Ø IF TRUE OUTPUT "TRUE"  
3Ø OUTPUT SENTENCE  
    THING OF PULL /LIST/  
    COLLECT REST OF /LIST/  
END
```

We can clean up the resulting list of contents by eliminating redundancies and "X"s.

```
TO CLEANUP /LIST/
1Ø TEST EMPTY /LIST/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø TEST EITHER
    IS FIRST /LIST/ "X"
    CP (FIRST /LIST/) (BUTFIRST /LIST/)
4Ø IF TRUE OUTPUT BUTFIRST /LIST/
5Ø OUTPUT SENTENCE
    FIRST /LIST/
    CLEANUP REST /LIST/
END
```

Again referring to the example from page 39:

```
+PRINT COLLECT /AREAS/
Ø 1 X X X 1 X
+PRINT CLEANUP COLLECT /AREAS/
Ø 1
```

Now we have the list of all join numbers of interest. We can use the existing procedure JOINLIST to generate the list of regions joined by any one of them. Thus,

```
TO TESTJOINS /ELEMENTS/ /REGION LIST/
1Ø MAKE "CONCLUSION REGIONS"
    SUBLIST /ELEMENTS/ /REGION LIST/
2Ø MAKE "NUMBERS" CLEANUP OF COLLECT
    /CONCLUSION REGIONS/
3Ø TEST CP "0" /NUMBERS/
4Ø IF TRUE OUTPUT "TRUE"
5Ø OUTPUT TRYJOINS /NUMBERS/ /CONCLUSION REGIONS/
    /REGION LIST/
END
```

TRYJOINS goes through the join numbers of interest, one by one.

```

TO TRYJOINS /NUMBERS/ /CONCLUSION REGIONS/ /REGION LIST/
1Ø TEST EMPTY /NUMBERS/           (Any numbers left to try?)
2Ø IF TRUE OUTPUT "FALSE"         (If not, we have failed)
3Ø MAKE "TRIAL" JOINLIST
    FIRST /NUMBERS/
    /REGION LIST/
    /EMPTY/                       (The last input for JOINLIST is
                                   the region name to be excluded.
                                   Here we have no such region)
4Ø TEST WITHINP /TRIAL/
    /CONCLUSION REGIONS/
5Ø IF TRUE OUTPUT "TRUE"
6Ø OUTPUT TRYJOINS (BUTFIRST
    /NUMBERS/) /CONCLUSION
    REGIONS/ /REGION LIST/
END

```

```

+PRINT TESTJOINS "COMPUTERS" /AREAS/
TRUE
+PRINT TESTJOINS "COMPUTERS MACHINES" /AREAS/
TRUE
+PRINT TESTJOINS "COMPUTERS NOT-MACHINES" /AREAS/
FALSE

```

Now that we have our two test "primitives", we can use them in the same manner we used the marking primitives in Section 5.

```

TO TESTCONCLUSION /STATEMENT/ /REGION LIST/
1Ø MAKE "A" DASH BUTFIRST LEFTS "ARE" /STATEMENT/
2Ø MAKE "B" DASH RIGHTS "ARE" /STATEMENT/
3Ø MAKE "ALL REGIONS" /REGION LIST/
4Ø DO WORD "TEST" (FIRST /STATEMENT/)
END

```

We again have four procedures based on the four statement forms. The conclusion, /STATEMENT/, is or is not valid as the appropriate combination of primitives outputs "TRUE" or "FALSE" so we write a little "translation" procedure

```

TO VALIDATE /P/
1Ø TEST IS /P/ "TRUE"
2Ø IF TRUE PRINT "VALID"
3Ø IF FALSE PRINT "NOT VALID"
END

```

Then,

```
TO TESTALL
10 VALIDATE BOTH
    TESTJOINS (SENTENCE /A/ /B/) /ALL REGIONS/
    TESTMARKS (SENTENCE /A/ WORD "NOT-" /B/) /ALL REGIONS/
END
```

```
TO TESTSOME
10 VALIDATE TESTJOINS (SENTENCE /A/ /B/) /ALL REGIONS/
END
```

```
TO TESTNO
10 VALIDATE TESTMARKS (SENTENCE /A/ /B/) /ALL REGIONS/
END
```

```
TO TESTNOT
10 VALIDATE TESTJOINS (SENTENCE BF BF BF BF /A/
    WORD "NOT-" /B/) /ALL REGIONS/
END
```

Again, using the two premisses on Page 39, "All computers are machines" "All machines are things that cannot think", and the equivalent diagram, indicated by region name list /AREAS/:

```
+TESTCONCLUSION "ALL THINGS THAT CANNOT THINK ARE COMPUTERS"
/AREAS/
NOT VALID
+TESTCONCLUSION "NOT ALL COMPUTERS ARE THINGS THAT CANNOT
THINK" /AREAS/
NOT VALID
+TESTCONCLUSION "SOME MACHINES ARE COMPUTERS" /AREAS/
VALID
```

7. More General Statement Forms

Syllogistic arguments rarely appear in exactly the forms presented above. Many arguments require some translation before they fit these standard forms. An example of such an argument is: "All my pets purr. Only cats purr. Therefore all my pets are cats." We first replace the active verb "purr" by the phrase "are things that purr", so that the argument now deals with class relations and is of the form:

$$\begin{array}{l} \text{All A are B} \\ \text{Only C are B} \\ \hline \therefore \text{All A are C} \end{array}$$

What about the second premiss? "Only C are B" means that nothing is a B when it is a C, that is, all B's are C's. We now have translated the argument into the standard syllogistic form:

$$\begin{array}{l} \text{All A are B} \\ \text{All B are C} \\ \hline \therefore \text{All A are C} \end{array}$$

Similarly, any statement of the form "Only A are B" can be translated into "All B are A". Another variation of the form "Only A are B" is "None but A are B". Again it means that every B is an A, i.e., "All B are A".

The following is a list of some common variations of the four basic forms. Others can be added.

All A are B

Only B are A
None but B are A
Every A is B
Each A is B
A is B
A is always B
No A are not B

Some A are B

Some B are A
Not all A are not B
A is sometimes B

No A are B

No B are A
It is not the case that A are B
A is not B
A is never B
A's are not B's
All A are not B
All B are not A

Not all A are B

Some A are not B
A is not always B

While many arguments can readily be translated into syllogistic form, there are others for which no such translation is appropriate. An example of such an argument is:

Either Smith lied or Jones was guilty.
Smith didn't lie.

∴ Jones was guilty.

In general, statements involving disjunctions (___or___) or conjunctions (___and___) are best treated by other means. The syllogistic form of argument does not allow for the combining of classes as in:

All cats are small.
All cats are animals.

∴ All cats are small animals.

Syllogistic reasoning also, for example, cannot account for the difference between the arguments:

$$\begin{array}{l} A \text{ precedes } B \\ B \text{ precedes } C \\ \hline \therefore A \text{ precedes } C \end{array}$$

$$\begin{array}{l} A \text{ likes } B \\ B \text{ likes } C \\ \hline \therefore A \text{ likes } C \end{array}$$

Let us proceed, therefore, limiting ourselves to statements equivalent to the four basic forms.

8. Generalizing Our Syllogism Tester

There are two major areas in which our abilities to accept non-standard inputs can be improved. They both appear in the traditional syllogism:

$$\begin{array}{l} \text{Socrates is a man} \\ \text{All men are mortal} \\ \hline \therefore \text{Socrates is mortal} \end{array}$$

That Socrates refers to a specific person and not a class of objects is an objection simply overcome by taking the class Socrates to comprise just that one object. This was dealt with in Section 1. But, there are two other more substantial difficulties. The first statement is not in one of the four standard forms. Thus, it will have to be translated into the appropriate one of the four forms before our syllogism testing procedures can be applied. Also, to a very literal-minded procedure, "a man" is quite different from "men". We must, in some way, teach the computer to look for such similarities which often imply that references to a single class have been made. Another example of such variants is "crying babies" and "babies who cry". This section will deal with partial resolutions of these two problems.

We first consider the extension of statement forms to include all those forms given in Section 7. To do this we make use of a special LOGO procedure MATCHP. MATCHP is part of a very general

pattern-matching set of programs written in LOGO. This set of programs is available on request from BBN. MATCHP takes two inputs, the trial and the pattern sought. If a match can be made, MATCHP outputs "TRUE" and, in case of failure, "FALSE". The pattern is specified both by literals and variables, the latter having a colon as their first character. Thus,

```
+MATCHP "ALL BUT GREEN MEN ARE PURPLE" "ALL :A ARE :B"
TRUE
+PRINT /:A/
BUT GREEN MEN
+PRINT /:B/
PURPLE
```

and

```
+MATCH "EGGS ARE OVAL" ":A :B"
TRUE
+PRINT /:A/
EGGS ARE OVAL
+PRINT /:B/
+
      (The result is the empty thing)
      (We see the matching goes from left to right)
```

We can therefore put all the statement forms of interest on a list, each statement form in the form of a pattern input for MATCHP. For example, we write ALL ___ ARE ___ as ALL :A ARE :B. A statement in question, then, can be compared with each pattern on the pattern list using MATCHP until we are successful in finding a match. We then know the form which fits the given statement. If, in addition, we make the THING of each pattern the corresponding one of the four basic statements, we also have our translation problem done.

Unfortunately, because of the difficulties inherent in English syntax, a given statement may fit into more than one pattern and usually such apparent ambiguity leads to disastrous results. For example, consider "All but men are women". This sentence

fits not only the pattern "All but ____ are ____", but also the pattern "All ____ are ____". Perhaps the simplest way out of this dilemma is to use the fact that if a statement fits two patterns, the correct fit (if either is correct) is given by the more restrictive of the two. Thus, if we order our patterns, the less restrictive following those which "include" them, the first matching pattern can always be taken as the correct one. With this in mind, we create a permanent "PATTERN LIST".

```
+MAKE "PATTERN LIST" "ONLY :B ARE :A , NONE BUT :B ARE :A , EVERY
:A IS :B , EACH :A IS :B , :A IS ALWAYS :B , NO :A ARE NOT :B ,
ALL :B ARE NOT :A , ALL :A ARE :B , SOME :A ARE NOT :B , SOME :B
ARE :A , NOT ALL :A ARE NOT :B , :A IS SOMETIMES :B , SOME :A ARE
:B , NO :B ARE :A , IT IS NOT TRUE THAT SOME :A ARE :B , IT IS
NOT THE CASE THAT SOME :A ARE :B , :A IS NEVER :B , :A ARE NOT :B
, ALL :A ARE NOT :B , NO :A ARE :B , :A IS NOT ALWAYS :B , NOT
ALL :A ARE :B , :A IS NOT :B , :A IS :B"
```

and

```
+MAKE "ONLY :B ARE :A" "ALL :A ARE :B"
+MAKE "NONE BUT :B ARE :A" "ALL :A ARE :B"
+MAKE "EVERY :A IS :B" "ALL :A ARE :B"
+MAKE "EACH :A IS :B" "ALL :A ARE :B"
+MAKE ":A IS ALWAYS :B" "ALL :A ARE :B"
+MAKE "NO :A ARE NOT :B" "ALL :A ARE :B"
+MAKE "ALL :A ARE NOT :B" "NO :A ARE :B"
+MAKE "ALL :A ARE :B" "ALL :A ARE :B"
+MAKE "SOME :A ARE NOT :B" "NOT ALL :A ARE :B"
+MAKE "SOME :B ARE :A" "SOME :A ARE :B"
+MAKE "NOT ALL :A ARE NOT :B" "SOME :A ARE :B"
+MAKE ":A IS SOMETIMES :B" "SOME :A ARE :B"
+MAKE "SOME :A ARE :B" "SOME :B ARE :A"
+MAKE "NO :B ARE :A" "NO :A ARE :B"
+MAKE "IT IS NOT TRUE THAT SOME :A ARE :B" "NO :A ARE :B"
+MAKE "IT IS NOT THE CASE THAT SOME :A ARE :B" "NO :A ARE :B"
+MAKE ":A IS NEVER :B" "NO :A ARE :B"
+MAKE ":A ARE NOT :B" "NO :A ARE :B"
+MAKE "ALL :A ARE NOT :B" "NO :A ARE :B"
+MAKE "ALL :B ARE NOT :A" "NO :A ARE :B"
+MAKE "NO :A ARE :B" "NO :A ARE :B"
+MAKE "SOME :A ARE NOT :B" "NOT ALL :A ARE :B"
+MAKE ":A IS NOT ALWAYS :B" "NOT ALL :A ARE :B"
+MAKE "NOT ALL :A ARE :B" "NOT ALL :A ARE :B"
+MAKE ":A IS NOT :B" "NO :A IS :B"
+MAKE ":A IS :B" "ALL :A ARE :B"
```

To find the pattern on /LIST/ corresponding to /STATEMENT/ we write:

```
TO FINDPATTERN /STATEMENT/ /LIST/
10 TEST EMPTY /LIST/
20 IF TRUE EXIT "NO PATTERN FOUND"
30 TEST MATCHP /STATEMENT/ (PULL /LIST/)
40 IF TRUE OUTPUT PULL /LIST/
50 OUTPUT FINDPATTERN /STATEMENT/ (REST /LIST/)
END
```

```
+PRINT FINDPATTERN "ALL ZILCHES ARE GRELBS"
ALL :A ARE :B
+PRINT /:A/
ZILCHES
+PRINT /:B/
GRELBS
```

We incorporate FINDPATTERN within a procedure TAKESTATEMENTS which generates both the list of /CLASSES/ and of /STATEMENTS/, eliciting and interpreting user responses. Lines 10 through 40 of ACCEPT will then be replaced by the three lines:

```
10 MAKE "CLASSES" /EMPTY/
20 MAKE "STATEMENTS" /EMPTY/
30 TAKESTATEMENTS
```

```
TO TAKESTATEMENTS
10 PRINT "PREMISS:"
20 MAKE "NEW" REQUEST
30 TEST EMPTY /NEW/
40 IF TRUE MAKE "CLASSES"
    DASHALL /CLASSES/
```

(DASHALL replaces the list of classes, separated by commas by "dashing" each class name and making a simple LOGO sentence out of them. This is the form that REGIONLIST, and hence ACCEPT, requires)

```
50 IF TRUE STOP
60 MAKE "FORM" FINDPATTERN /NEW/
    /PATTERN LIST/
70 ADDON /:A/ /CLASSES/
80 ADDON /:B/ /CLASSES/
```

(Adds /:A/, /:B/ to /CLASSES/ if they are not there already)

```

90 TEST IS /FORM/ THING OF /FORM/ (Is the basic pattern corre-
100 IF FALSE sponding to /FORM/, /FORM/
    PRINT SENTENCE SENTENCE itself? If so, we have a basic
    SENTENCE SENTENCE SENTENCE pattern.)
    "I UNDERSTAND,"
    THING /FORM/
    ", WHERE :A IS"
    /:A/
    ", AND :B IS"
    /:B/
110 MAKE "STATEMENTS" SENTENCE (Puts current /:A/ and /:B/
    SENTENCE into THING OF /FORM/, obtaining
    /STATEMENTS/ the standard form of our
    SUBSTITUTE THING OF /FORM/ premiss)
    " "
120 TAKESTATEMENTS
END

```

And the subprocedures are quite simple:

```

TO ADDON /EL/ /LIST/
1Ø TEST ONP /EL/ /LIST/
2Ø IF TRUE STOP
3Ø MAKE /LIST/ SENTENCE SENTENCE
    THING OF /LIST/
    /EL/
    ","
END

```

(ONP was used in Section 6.)

(The comma seems to be in the wrong place but tracing the procedure shows it isn't)

```

TO SUBSTITUTE /SENT/
1Ø TEST EITHER
    IS FIRST /SENT/ ":A"
    IS FIRST /SENT/ ":B"
2Ø IF TRUE OUTPUT SENTENCE
    THING FIRST /SENT/
    SUBSTITUTE (BUTFIRST /SENT/)
3Ø OUTPUT SENTENCE
    FIRST /SENT/
    SUBSTITUTE (BUTFIRST /SENT/)
END

```

```

TO DASHALL /LIST/
1Ø TEST EMPTY /LIST/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø OUTPUT SENTENCE
    DASH PULL /LIST/
    DASHALL (REST /LIST/)
END

```

With TAKESTATEMENT incorporated in ACCEPT, as described above:

```

+ACCEPT
PREMISS:
*SOCRATES IS A MAN
I UNDERSTAND, ALL :A ARE :B , WHERE :A IS SOCRATES ,
AND :B IS A MAN
PREMISS:
*ALL MEN ARE MORTAL
PREMISS:
*           (carriage return)
NAME OF REGION LIST:
*GREEKS
+PRINT /STATEMENT/
ALL SOCRATES ARE A MAN , ALL MEN ARE MORTAL
+PRINT /CLASSES/
SOCRATES , A MAN , MEN , MORTAL

```

We have achieved our first goal, extending ACCEPT to *accept* a wide variety of statement forms. But, as PRINT /CLASSES/ shows us in the above example, we cannot yet *deal with* such cases. Let us treat "similarity" by first writing a top level procedure. This is often a useful approach because we see exactly what subprocedures are needed, instead of having to combine previously written subprocedures in a possibly clumsy way: CHECK /CLASS/ /LIST/ tries to find a member of /LIST/ which is "similar" to /CLASS/. If it is found, we are given a choice between the similar object and /CLASS/, CHECK outputting the *user's choice*. ("SPEAR", after all is not the same as "PEARS" even though the words are very similar, both in the letters they contain and the order in which the letters occur.)

```

TO CHECK /CLASS/ /LIST/
1Ø TEST EMPTY /LIST/
2Ø IF TRUE OUTPUT /CLASS/
3Ø TEST SIMILARP /CLASS/ (PULL /LIST/) (SIMILARP is our similarity
                                         predicate, as yet undefined)
4Ø IF TRUE TEST IS                      (Is the user's choice the
    CHOICE /CLASS/ (PULL /LIST/)        similar object, if so
    PULL /LIST/                          output it)
5Ø OUTPUT CHECK /CLASS/ (REST /LIST/)
END

```

Note that we are able to specify this procedure, even before we know what conditions for similarity will be used. This is a useful feature of this manner of programming. The higher level programs can be made, independent of most specific choices.

CHOICE is simple:

```

TO CHOICE /A/ /B/
1Ø PRINT SENTENCE SENTENCE SENTENCE
    "DO YOU MEAN"
    /A/
    "TO BE THE SAME AS"
    /B/
    "?"
2Ø TEST IS REQUEST "YES"
3Ø IF TRUE OUTPUT /B/
4Ø OUTPUT /A/
END

```

```

+PRINT CHOICE "HEADS" "TAILS"
DO YOU MEAN HEADS TO BE THE SAME AS TAILS ?
*YES
TAILS

```

We are left with the task of writing SIMILARP /A/ /B/ which incorporates our choice of similarity criterion. After some investigation, it turns out that simply comparing the sets of letters making up /A/ and /B/ is a simple but fairly satisfactory approach. We arbitrarily say that if /A/ and /B/ contain more

letters in common than half the length of the shorter input, or if the number of letters in common exceeds 3, /A/ /B/ are "similar". If /A/ and /B/ are identical, we do not regard them as similar -- why use CHOICE when its two inputs are the same? Our similarity criterion gives us many false matches, e.g., "PEARS" and "SPEAR" as above. In fact, very few criteria will distinguish the two and this is why the final choice is put by the computer to the user.

To compare /A/ /B/, we first SQUASH them into LOGO words, write COMPARE1 to count the number of letters in common and SIMILARP uses these two to check for similarity.

```
TO SQUASH /A/
10 TEST EMPTY /A/
20 IF TRUE OUTPUT /EMPTY/
30 OUTPUT WORD (FIRST /A/) (SQUASH BUTFIRST /A/)
END
```

```
TO COMPARE1 /A/ /B/
10 TEST EITHER
    EMPTY /A/
    EMPTY /B/
20 IF TRUE OUTPUT 0
30 TEST CP (FIRST /A/) /B/      (CP is used earlier to check if
                                its first input is contained
                                within its second one)
40 IF TRUE OUTPUT SUM
    1
    COMPARE1 (BUTFIRST /A/) DELETE (FIRST /A/) /B/
50 OUTPUT COMPARE1 (BUTFIRST /A/) /B/
END
```

```
+PRINT COMPARE1 (SQUASH "FOUR YELLOW YAKS") (SQUASH "CUCUMBERS")
4
```

```

TO SIMILARP /A/ /B/
10 TEST IS /A/ /B/
20 IF TRUE OUTPUT "FALSE"
30 MAKE "A" SQUASH /A/
40 MAKE "B" SQUASH /B/
50 MAKE "COMMON" COMPARE1 /A/ /B/
60 MAKE "LESS COUNT" MINIMUM (COUNT /A/) (COUNT /B/)
70 OUTPUT EITHER
    GREATERP /COMMON/ 3
    GREATERP /COMMON/ (QUOTIENT /LESS COUNT/ 2)
END

```

```

+PRINT SIMILARP "A GREEN MONSTER" "GREEN MONSTERS"
TRUE
+PRINT CHECK "A MAN" "WOMEN MEN CHILDREN"
DO YOU MEAN A MAN TO BE THE SAME AS MEN ?
*YES
MEN

```

Now that we have completed CHECK, it is an easy matter to incorporate it in our existing syllogism testing procedures. ACCEPT need not be modified directly, since in this section we wrote TAKESTATEMENTS to accept statements and create a list of classes, replacing lines 10 through 40 of ACCEPT by new ones. In TAKESTATEMENTS itself we now have lines 70 and 80 for extending the class list:

```

70 ADDON /:A/ /CLASSES/
80 ADDON /:B/ /CLASSES/

```

and we need only modify them slightly

```

70 ADDON (CHECK /:A/ /CLASSES/) /CLASSES/
80 ADDON (CHECK /:B/ /CLASSES/) /CLASSES/

```

This change finishes the modernization of our premiss accepting procedure ACCEPT. We must do the same for TESTCONCLUSIONS given in Section 6. We also have not yet incorporated in it our generalization of input statement procedures. Let us rewrite the entire procedure, making the changes implied by the work of this section.

```

TO TESTCONCLUSION /STATEMENT/ /REGION LIST/
10 MAKE "A" BUTFIRST LEFTS "ARE" /STATEMENT/
20 MAKE "A" DASH (CHECK /A/ /CLASSES/)
30 MAKE "B" RIGHTS "ARE" /STATEMENT/
40 MAKE "B" DASH (CHECK /B/ /CLASSES/)
50 MAKE "STATEMENT" SUBSTITUTE (FINDPATTERN
    /STATEMENT/ /PATTERN LIST/)
60 DO WORD "TEST" FIRST /STATEMENT/
END

```

Essentially, the changes we have made were adding lines 20 and 40 to CHECK the classes and line 50 to convert the statement into one of the four standard forms.

We are finished. It remains only to try out our completed and modernized procedure package:

```

←ACCEPT
PREMISS:
*SOCRATES IS A MAN
I UNDERSTAND, ALL :A ARE :B , WHERE :A IS SOCRATES ,
AND :B IS A MAN
PREMISS:
*ALL MEN ARE MORTAL
DO YOU MEAN MEN TO BE THE SAME AS A MAN?
*YES
PREMISS:
*      (carriage return)
NAME OF REGION LIST:
*GREEKS
←TESTCONCLUSION "SOCRATES IS MORTAL" /GREEKS/
VALID
←

```

PROBLEMS

LOGO Teaching Sequence on Logic

1. In the text, NICEPRINT is used to list the region names together with their contents. Write a procedure CARROLL, which, if given a list of region names involving three classes, gives us the corresponding Lewis Carroll diagram. Two possibly useful comments are:

- 1) It is best to first try to draw Lewis Carroll diagrams without labelling regions according to the classes they represent.
- 2) The straight lines in the diagram are always unchanged, except those parts where joins can be placed.

2. Extend the procedure TESTCONCLUSION, used for testing the validity of a conclusion, to one which tests conclusions that are the conjunctions of several statements. (In such a case, each statement in the conclusion must follow from the premisses separately.)

3. The list of patterns given in Section 8 must be in proper order. More specific patterns must precede less restrictive ones. Write a procedure which inputs a list of patterns and outputs them reordered as necessary.

4. If a premiss is entered which results in a contradiction, MARKOFF will often spoil the premiss diagram. This is due to the way MARKOFF works -- it marks off regions until the contradiction is found in the form of a region containing an "0". MARKOFF then stops, but the previously marked x's remain. Rewrite MARKOFF to overcome this difficulty.

5. Any statement that contains the word "are" more than once poses special difficulties for our premiss processing procedures, since a statement like "all people who are fat are big eaters" is apparently ambiguous.

A) Write a procedure which checks for this eventuality and indicates this by printing a suitable message.

B) Write a procedure which correctly interprets such statements. Use the fact that in statements containing two "ares", the "are" which is part of a class name is usually preceded by "who", "that", or "which".

6. Write a procedure which, given a set of premisses, outputs the set of all valid conclusions. To do this, note there are four basic statement forms and the set of all classes is that given by the premisses. The set of all possible conclusions, invalid as well as valid, is given by putting each pair of classes in each of the statement forms.

7. A) Write a procedure which, given a set of classes, randomly generates premisses using them.

B) Combine this procedure with the one described in Problem 6 to make a complete valid syllogism generator.

C) Use the result of Part B) in a procedure which gives a student syllogisms, asks whether they are valid, and evaluates his answers.

8. Use the procedure written for Problem 6 and the CHECK procedure written in Section 8 to write a "syllogism helper". This procedure, if given an invalid syllogism, gives the user the valid syllogism which is nearest to it in terms of the words contained in it.

NOTE: "Symbolic Logic and the Game of Logic" by Lewis Carroll, reprinted by Dover Publications, 1958, is a rich source for all sorts of syllogisms.

Volume 1, Part 3

LOGO REFERENCE MANUAL

The LOGO Project

NSF-C 615

Wallace Feurzeig

George Lukas

Richard Grant

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

CONTENTS

	Page
1. A Look at LOGO	1
2. The LOGO Language	3
2.0 Mechanics	3
2.1 LOGO Objects and the PRINT Command	3
2.2 Constructing a Procedure	4
2.3 Procedures With Inputs	5
2.4 LOGO Operations	6
2.5 Composition	8
2.6 Procedures With Outputs	11
2.7 Naming	12
2.8 Conditional Operations	15
2.9 Recursion	17
3. Program Manipulation	20
3.1 Editing	20
3.2 Abbreviating	23
3.3 The User Workspace	24
3.4 Filing	25
3.5 Debugging	27
3.6 Interrupting Execution	30
4. Additional Commands and Operations	31
4.1 Formatting	31
4.2 Interactive Programs	32
4.3 More Arithmetic	34
4.4 Local and Global Names	35
4.5 Automatic Program Generation - An Advanced Feature	36
4.6 Other Ways to Terminate a Procedure	38
4.7 Miscellany	39
5. Glossary and Index	41

LOGO REFERENCE MANUAL

1. A Look at LOGO

We introduce LOGO by writing several small procedures. The following examples serve to show what LOGO "looks like". Several features are used without definition or even explanation, where we think their meanings are clear from context. All of LOGO is comprehensively described in later sections.

LOGO, as an interpretive language, can execute single commands directly. Thus,

```
+PRINT SUM OF 2 AND 2          (The user's typing is underlined)
4
```

But, the most important feature of LOGO is that such commands can be incorporated in user-written procedures. The definition of any procedure results in an object which is treated just like any primitive. Thus, in a very real sense, as the user writes his own procedures, he is gradually extending the basic language to more exactly fill his needs.

A very simple (although by no means simplest) procedure, for example, prints the double of its input.

```
TO DOUBLE /N/
  10 PRINT SUM OF /N/ AND /N/
END
```

This procedure, DOUBLE, is now "part" of LOGO.

```
+DOUBLE 123
246
+DOUBLE WORD OF 1 AND 1
22
```

If the concatenation of DOUBLE with other procedures is desired, DOUBLE should OUTPUT rather than PRINT its results; OUTPUT meaning that the result is given to the calling procedure. The modified procedure is:

```
TO DOUBLE /N/
  10 OUTPUT SUM OF /N/ AND /N/
END
```

This new version of DOUBLE can be used in direct commands,

```
+PRINT DOUBLE DOUBLE DOUBLE 3
24
```

```

TO QUADRUPLE /N/
10 OUTPUT DOUBLE OF DOUBLE OF /N/
END

```

An extension of this LOGO facility for using procedures in defining other procedures is its ability to handle recursive procedure definitions. The recursion can be a linear one, which is equivalent to iteration, as in the following procedure which calculates the factorial function: $n! \equiv n \cdot (n-1) \cdots 2 \cdot 1$.

```
+PRINT FACTORIAL OF 100  
9332621544394415268169923885626670049071596826438162146859296389521  
7599993229915608941463976156518286253697920827223758251185210916864  
00000000000000000000000000000000
```

```

TO REVERSE /STRING/
10 TEST IS /STRING/ /EMPTY/      (The reverse of the empty string
20 IF TRUE OUTPUT /EMPTY/         is the empty string.)
30 OUTPUT WORD OF LAST OF /STRING/ (The reverse of the nonempty
    AND (REVERSE OF BUTLAST OF    string is the string formed by
    /STRING/)                     following the last character by
END                                the reverse of the rest of the
                                  string.)

```

←PRINT REVERSE OF "JABBERWOCKY"
YKCOWREBBAJ

2. The LOGO Language

2.0 Mechanics

To start using LOGO, a person must establish communication with the computer and specify that he wishes to work with LOGO. The conventions for doing this vary from system to system and are, therefore, outside the bounds of this manual.

To indicate that it is ready for use, LOGO types a back-arrow (+) either immediately or in response to the pressing of the key labeled "RETURN". This means that LOGO is ready to receive an instruction line. After the user types in the desired line, he again presses the RETURN key. This action returns the carriage and gives the command line to LOGO for execution. The LINE FEED key also returns the carriage, but does not cause command execution; thus an arbitrarily long instruction can be entered.

The typing of a line may be aborted at any time by pressing the key labeled "RUB OUT".

The key labeled "BREAK" is used to stop the execution of the current instruction and return control to the user.

The use of LOGO is terminated by typing the command "GOODBYE" or its abbreviation "GB".

Other LOGO system facilities, such as erasing, editing, and filing, are discussed in the section on program manipulation.

2.1 LOGO Objects and the PRINT Command

LOGO contains two kinds of objects, LOGO words and LOGO sentences. A LOGO word is an arbitrary string of printing teletype characters excluding quote marks. A LOGO sentence is an arbitrary string of LOGO words, separated by spaces. If a user types multiple spaces between words of a LOGO sentence, all but one are automatically eliminated.

To simply print a LOGO object on the teletype, the PRINT command is used.

```
+PRINT "Y!!RCHN4LK"  
Y!!RCHN4LK  
+PRINT "32425"  
32425
```

(a LOGO word)

(a LOGO word)

+PRINT "HI THERE LOGO"

(a LOGO sentence)

HI THERE LOGO

+PRINT ""

(the "empty" object)

(prints out an empty line)

←

LOGO objects are delimited by quote marks. If a word is not so delimited, it is taken as a LOGO command or operation. Integer objects are an exception -- for these, quotes are optional.

+PRINT 123450

123450

2.2 Constructing a Procedure

Using the "built-in" LOGO commands and operations, such as PRINT, we construct procedures either to manipulate LOGO objects or to produce some desired external effects. For example, a LOGO procedure for drawing a small triangle out of + marks is:

+TO TRIANGLE

>5 PRINT "+"

>10 PRINT "++"

>32 PRINT "+++"

>40 PRINT "++++"

>END

←

To get LOGO to perform this procedure, we merely type the procedure name:

+TRIANGLE

+

++

+++

++++

←

A procedure definition includes three different kinds of lines. The first line of the definition is called the title line. It begins with the LOGO command TO which indicates to LOGO that we are writing a procedure whose name immediately follows. (TO is not part of the procedure name.) The procedure name must not be a command or operation currently used by LOGO. (This excludes built-in commands like PRINT as well as user-defined procedures in the current workspace.) And, like the LOGO built-in commands and operations, procedure names must not contain quotes or slashes.

After reading the title line, LOGO types back the wedge mark (>). This indicates that it is ready for the next line of the definition. The actions of the procedure are specified by the instruction lines immediately following the title line. Each of these is prefaced by an integer, a *line number*. Following the line number is the instruction itself, such as a LOGO command. The instructions are not executed during this definition phase -- they are merely entered, checked for local syntax errors, and stored away for subsequent execution. Later, when the procedure is to be performed, LOGO will execute these instruction lines in increasing order, by line number, even if they are not set down in such order. The END line indicates the close of the definition.

2.3 Procedures With Inputs

The procedure TRIANGLE always has the same effect. This contrasts with the built-in PRINT command whose effect varies with its input. As well as procedures with no inputs, like TRIANGLE, we can write procedures that have inputs, for example the following one:

```
TO PSYCH /ANYTHING/
1Ø PRINT "WHY DO YOU SAY"
2Ø PRINT /ANYTHING/
3Ø PRINT "?"
END
```

To use PSYCH, we must give it one input:

```
+PSYCH "YOUR NAME IS ELIZA"
WHY DO YOU SAY
YOUR NAME IS ELIZA
?
+
```

```
+PSYCH "2+2=4"
WHY DO YOU SAY
2+2=4
?
+
```

By adding a word or sentence enclosed within slashes -- a placeholder or *dummy input* -- to its title line, we denote that PSYCH requires one input. When PSYCH is used, LOGO replaces the dummy input, wherever it occurs in the procedure definition, by the desired input (such as "2+2=4").

We write and use procedures with more than one input along analogous lines. For example:

```
TO REVERSE /FIRST/ /SECOND/ /THIRD/
1Ø PRINT /THIRD/
2Ø PRINT /SECOND/
3Ø PRINT /FIRST/
END
```

REVERSE needs three inputs.

```
+REVERSE "A" "B" "C"
```

```
C
B
A
```

```
+REVERSE "ABC"
```

```
THERE ARE 2 INPUTS MISSING FOR REVERSE
+
```

In the second example, we typed only one input. (When an illegal instruction is attempted, LOGO responds with an appropriate error message. A list of such messages is included in Section 3.5.)

2.4 LOGO Operations

A LOGO *command* (such as PRINT, TO, END, and GOODBYE) always results in some external action. For example, PRINT causes its input to be printed on the teletype. An *operation*, however, *passes on* a LOGO object for further use within LOGO. For example, the LOGO operation SUM requires two inputs (integers), and *outputs* their sum. This new object can either be used as an input for PRINT, if we wish to have it typed, or as an input to a LOGO operation or procedure.

```
+PRINT PRODUCT OF 2 AND 7
```

```
14
```

```
+PRINT PRODUCT OF 2 AND (SUM OF 3 AND 4)
```

```
14
```

```
+
```

Here, the output of SUM, 7, is one of the two inputs to the operation PRODUCT which, in turn, passes its output, 14, to PRINT. [OF and AND are completely optional and are used only to improve clarity and readability of expressions for the user. OF may be used after any command, operation, or procedure name. AND may be used between inputs. Balanced parentheses also may be used to improve readability. (See the section following on composition.)]

LOGO includes operations of different kinds, for example, those for concatenation and decomposition given below.

Concatenation

WORD concatenates two given words, its inputs, to form a new word as its output. SENTENCE concatenates its two inputs, either words or sentences, to form a new sentence as its output.

```
+PRINT WORD OF "UP" AND "DOWN"
UPDOWN
+PRINT SENTENCE OF "UP" AND "DOWN"
UP DOWN
+PRINT SENTENCE OF "GO MAN" AND "GO"
GO MAN GO
+PRINT WORD OF "GO MAN" AND "GO"
THE INPUTS TO WORD MUST NOT BE SENTENCES
←
```

Decomposition

FIRST outputs the first character of its input, if the input is a word (or the first word, if the input is a sentence). BUTFIRST outputs everything but the first of its input (i.e., the second-through-last characters of word inputs, or the second-through-last words of sentence inputs). LAST and BUTLAST are defined similarly.

```
+PRINT FIRST OF "CAT"
C
+PRINT BUTFIRST OF "CAT"
AT
+PRINT LAST OF "CAT"
T
+PRINT BUTLAST OF "CAT"
CA
+PRINT FIRST OF "FEE FIE FOE FUM"
FEE
+PRINT BUTFIRST OF "FEE FIE FOE FUM"
FIE FOE FUM
+PRINT LAST OF "A"
A
+PRINT BUTLAST OF "A"

```

(LOGO prints the empty word)

←

The output of FIRST or LAST is always a LOGO word. The output of BUTFIRST or BUTLAST, however, is of the same type (i.e., LOGO word or sentence) as its input. This means that a single word can sometimes be a LOGO sentence. Thus, the output of BUTFIRST OF "THE CAT" is the LOGO one-word sentence "CAT".

```
+PRINT FIRST OF BUTFIRST OF "THE CAT"
CAT
←
```

Arithmetic

The operations SUM, DIFFERENCE, PRODUCT, and QUOTIENT take two inputs, which must be integers, and output the designated arithmetic result. The operation RANDOM has no input and outputs a digit between 0 and 9 generated in a pseudo-random manner.

```
+PRINT SUM OF 2 AND 98
100
+PRINT DIFFERENCE OF 2 AND 98
-96
+PRINT PRODUCT OF 17 AND 5
85
+PRINT QUOTIENT OF 17 AND 5
3 (the integer quotient)
+PRINT RANDOM
8
+PRINT RANDOM
2
←
```

Other LOGO "built-in" operations are described later.

2.5 Composition

The inputs of LOGO operations do not have to be literal LOGO objects like "CAT" or "39". Instead, as we saw, they may be outputs of other operations. The following examples illustrate how operations can be composed or "chained".

```
+PRINT SENTENCE OF "A B" AND WORD OF "CD" AND "E"
A B CDE
+PRINT LAST OF (FIRST OF "THE CAT")
E
+PRINT DIFFERENCE OF (WORD OF 11 AND 22) AND 122
1000
←
```

Let's consider how LOGO scans and executes these composite instructions by considering the first example above. LOGO scans an instruction line word-by-word from left to right. During the scanning process it identifies each element and notes its type. Thus:

<u>ELEMENT</u>	<u>TYPE</u>
(1) PRINT	Command, needs one input
(2) SENTENCE	Operation, needs two inputs
(3) OF	"Noise" word
(4) "A B"	LOGO sentence literal
(5) AND	Noise word
(6) WORD	Operation, needs two inputs
(7) OF	Noise word
(8) "CD"	LOGO word literal
(9) AND	Noise word
(10) "E"	LOGO word literal

After the input line is scanned, LOGO returns to the beginning of the line and *executes* it as follows:

- (1) Fetch the first element PRINT. Its one input remains to be found: fetch the next element.
- (2) PRINT's one input will be the output of SENTENCE. SENTENCE, in turn, needs two inputs: fetch the next element.
- (3) OF is a noise word, legal in this context. Ignore it. We are still looking for two inputs.
- (4) "A B" is an acceptable first input for SENTENCE. The second input needs to be found for SENTENCE: fetch the next element.
- (5) AND is a noise word, legal in this context. Ignore it. One input remains to be found for SENTENCE.
- (6) SENTENCE's second input will be the output of WORD. WORD, in turn, needs two inputs -- LOGO words: fetch the next element.
- (7) OF is a noise word, legal here. Ignore it. We are still looking for two inputs for WORD.
- (8) "CD" is an acceptable first input for WORD: fetch the next element.
- (9) AND is a noise word, legal here. Ignore it. We are still looking for the second input for WORD.

- (10) "E" is an acceptable second input for WORD. There are no further inputs being sought.
- (11) WORD outputs "CDE" to SENTENCE as its second input.
- (12) SENTENCE outputs "A B CDE" to PRINT as its one input.
- (13) PRINT causes A B CDE to be printed on the teletype page.
- (14) The processing of the line is complete. LOGO returns control to the user.

This process can be regarded perhaps more conveniently as a successive simplification of the original line as the outputs of various operations are determined. As an illustration, consider the following example:

```
+PRINT PRODUCT OF SUM OF 2 AND 3 AND DIFFERENCE OF 4 AND 5 AND 6
```

Unlike before, we immediately remove all noise words, thus obtaining:

```
+PRINT PRODUCT SUM 2 3 DIFFERENCE 4 5 6
```

We again go word-by-word from left to right. Finding upon reaching the 3 that we can complete the execution of SUM, we do so, obtaining:

```
+PRINT PRODUCT 5 DIFFERENCE 4 5 6
```

This execution of SUM does not permit any previously encountered operations to be executed so we keep on going to the right. We find that we can next complete the execution of DIFFERENCE. Doing this, we obtain:

```
+PRINT PRODUCT 5 -1 6
```

Again, looking back at previously encountered operations, we find that PRODUCT can now be executed. This results in:

```
+PRINT -5 6
```

Looking back once more, we find we can execute PRINT. We do so, thereby obtaining the printout -5. Since the first command on the original line has been executed, we deem the execution of the line to be complete. There is, however, something remaining on the line and this causes LOGO to print the error message:

```
"6" IS EXTRA
```

LOGO then returns control to the user.

Parentheses

Parentheses may be used in instructions to set off any expression that is, or can be used as, a LOGO input. The following examples of such use are valid:

```
+PRINT SUM (2) 3
+PRINT SUM 2 (3)
+PRINT (SUM 2 3)
+(PRINT SUM 2 3)
```

All these instructions cause the printout 5.

The following examples show some invalid uses of parentheses.

```
+PRINT (SUM) 2 3
+PRINT (SUM 2) 3
+PRINT SUM (2 3)
```

None of the above parenthesized expressions are meaningful LOGO inputs. The attempted execution of these lines will cause LOGO to print error messages such as MATCHING (? and MISSING)?.

Two infix expressions such as

$2 + (3 \times 4)$ and $(2 + 3) \times 4$

whose operations and inputs occur in the same order, but which are differently parenthesized, yield different results. Thus, the use of parentheses can change the order of evaluation of infix expressions. This is not so with LOGO prefix expressions. So long as parentheses are used correctly -- that is, to enclose inputs or possible inputs -- they cannot change the result of evaluating any LOGO prefix expression.

2.6 Procedures With Outputs

LOGO operations, by definition, output when they are executed. LOGO procedures can be written which output in just the same way as operations. To do this, the one-input LOGO command OUTPUT is used. Its use is illustrated:

```
TO DUBBLE /SOMETHING/
  10 OUTPUT WORD OF /SOMETHING/ AND /SOMETHING/
END
```

```
+PRINT DUBBLE "TROUBLE"
TROUBLETROUBLE
```

Note that, as in the case of built-in operations like FIRST or SUM, we need to preface DUBBLE with PRINT to cause the output of DUBBLE to be typed. In line 10 of DUBBLE, OUTPUT causes the result of execution of its line to be passed back as an input to the PRINT command which called DUBBLE. The LOGO command OUTPUT takes a single input and passes this back as an input to the command, operation, or procedure which called the present procedure. OUTPUT then causes resumption of the execution of the line to which it returned its input. At this point the outputting procedure effectively vanishes. Clearly, the command OUTPUT can only be used within a procedure.

User-defined procedures which output can be composed in the same way as built-in operations. Thus:

```
+PRINT SUM OF DUBBLE 3 AND 4
37
+PRINT DUBBLE DUBBLE DUBBLE "B"
BBBBBBBB

+TO QUADRUBBLE /STUFF/
>10 OUTPUT DUBBLE OF DUBBLE OF /STUFF/
>20 PRINT "I AM ALL DONE"
>END
QUADRUBBLE DEFINED
+PRINT QUADRUBBLE OF "NO"
NONONONO
```

In the last example note that the command in line 20 was not executed. The OUTPUT in line 10 terminated the execution of QUADRUBBLE as it passed back "NONONONO" to PRINT. Thus we now have two means of terminating a procedure, either by the END command or the OUTPUT command.

2.7 Naming

A LOGO object can be given a name by use of the MAKE command. MAKE takes two inputs and makes the first the name of the second. Thus:

```
+MAKE "DIGIT" "TOE"
```

We say that "DIGIT" is the *name* of "TOE" and that "TOE" is the *thing* of "DIGIT". To retrieve the thing of a name, we use the one-input LOGO operation THING. Thus:

```
+PRINT THING OF "DIGIT"
TOE
```

So that a name refers to precisely one thing at any time, MAKE replaces the old thing of the name by the given new thing, the second input of MAKE. For example:

```
+MAKE (WORD OF "DIG" AND "IT") (SENTENCE OF "BIG" AND "TOE")  
+PRINT THING OF "DIGIT"  
BIG TOE
```

Note that the previous thing of "DIGIT", "TOE", has been replaced by "BIG TOE".

Initially, the empty thing, "", is taken as the thing of any LOGO name. Therefore:

```
+PRINT THING OF "ABRACADABRA"
```

←

(The thing "" has been printed.)

For introductory purposes, an extended form of MAKE is available. This is used by pressing the return key directly after typing in the word MAKE. LOGO will then type NAME: and wait for the user to type in the desired name and press the return key. LOGO then responds by typing THING:. After the user types the desired thing and presses the return key, the command is executed in exactly the same way as the standard form. Thus:

```
+MAKE  
  NAME:  "1"  
  THING: "2"
```

```
+PRINT THING OF "1"  
2  
←
```

Because of the special importance of names, LOGO provides a shorthand notation for the operation THING. /IRVING/ means precisely the same as THING of "IRVING". Thus:

```
+PRINT SUM OF /1/ AND /1/  
4  
←
```

We have already encountered the use of slashes as delimiters for LOGO dummy inputs within procedure definitions. For example:

```

TO PYTHAGORAS /#/ /##/
10 PRINT SUM (PRODUCT /#/ /#/ ) (PRODUCT /##/ /##/)
END

```

When a procedure is executed, the occurrence of slashed objects in the title line results in implicit MAKE executions. Each is given as its thing the actual input in the corresponding position following the procedure name. For example, typing the command PYTHAGORAS 3 4 results in the name "#" being given the thing "3" and the name "##" being given the thing "4". /.../ means THING OF "..." whether the assignment was made by a procedure call or a MAKE command.

The names made by implicit MAKES as a result of the execution of a title line have a special status. These names vanish from the list of names currently known to LOGO upon termination of the procedure. This is true even if the initial assignment of a thing to a dummy name is changed by a MAKE within the procedure.

```

TO SHOW-OFF /M/ /N/
10 PRINT SENTENCE OF "M IS" /M/
20 PRINT SENTENCE OF "N IS" /N/
END

```

```

+SHOW-OFF "TURTLE" "EGGS"
M IS TURTLE
N IS EGGS
+PRINT /M/

+PRINT /N/

+

```

If a dummy name is already known to LOGO, LOGO sets up a special version of the name expressly for use within the procedure. As before, each dummy name is given as its thing the corresponding input. Thus:

```

+MAKE "M" "IRVING"
+SHOW-OFF "EGGS" AND "TURTLE"
M IS EGGS
N IS TURTLE
+PRINT /M/
IRVING

+

```

Such special treatment is given to a dummy name independently in each procedure where it occurs. Thus:

```

TO SHO /A/
1Ø PRINT SENTENCE "/A/ IS" /A/
2Ø SHOO (BUTFIRST OF /A/)
3Ø PRINT SENTENCE "/A/ IS" /A/
END

```

```

TO SHOO /A/
1Ø PRINT SENTENCE "/A/ IS" /A/
2Ø SHOOO (BUTFIRST OF /A/)
3Ø PRINT SENTENCE "/A/ IS" /A/
END

```

```

TO SHOOO /A/
1Ø PRINT SENTENCE "/A/ IS" /A/
END

```

```

←SHO "START"
/A/ IS START

```

```

/A/ IS TART

```

```

/A/ IS ART

```

```

/A/ IS TART

```

```

/A/ IS START

```

```

←

```

(Printed by SHO. At this point there is only one /A/)

(Printed by SHOO. Now there are two versions of /A/)

(Printed by SHOOO. Now there are three versions of /A/)

(Printed by SHOO. Two versions of /A/ remain since SHOOO has finished)

(Printed by SHO. Only one version of /A/ is left)

(All versions of /A/ have vanished)

2.8 Conditional Operations

LOGO includes a number of operations called predicates, whose outputs are one of the words "TRUE" or "FALSE". NUMBERP is the LOGO number predicate.

```

←PRINT NUMBERP OF "777"

```

```

TRUE

```

```

←PRINT NUMBERP OF "SEVEN"

```

```

FALSE

```

```

←PRINT NUMBERP OF WORD OF 2 AND 2

```

```

TRUE

```

```

←

```

WORDP and SENTENCEP are the LOGO word predicate and LOGO sentence predicate:

```

+PRINT WORDP OF 711
TRUE
+PRINT WORDP OF "WORDS WORDS WORDS"
FALSE
+PRINT SENTENCEP OF "I C A R U S"
TRUE
←

```

Two useful predicates requiring two inputs are the identity predicate, IS, and the predicate GREATERP for comparing two numbers. IS outputs "TRUE" if its inputs are the same, and "FALSE" if they are different; GREATERP outputs "TRUE" if its first input is strictly greater than its second input, and "FALSE" otherwise.

```

+PRINT IS SUM OF 2 AND 2 "4"
TRUE
+PRINT IS SUM OF 2 AND 2 "04"
FALSE
+PRINT GREATERP 2 2
FALSE
←

```

Some predicates require one of the words "TRUE" or "FALSE" as inputs. These include the conjunctive predicate BOTH and the disjunctive predicate EITHER. BOTH outputs "TRUE" if both of its inputs are "TRUE"; EITHER outputs "TRUE" if either one or the other or both of its inputs are "TRUE".

```

+PRINT BOTH (IS WORD OF "2" AND "2" "22") AND (NUMBERP OF "SEVEN")
FALSE
+PRINT EITHER (GREATERP 2 3) AND (WORDP OF "FALSE")
TRUE
←

```

A predicate's output can be tested for "TRUE" or "FALSE" and subsequent instruction execution can be made to depend upon the result. LOGO provides a command, TEST, to facilitate such tests.

+TEST IS "1" "1"	(Clearly true)
+IF FALSE PRINT "1 IS NOT 1"	(So this line is not executed)
+IF TRUE PRINT "IT IS!"	(But, this one is)
IT IS!	
+IF TRUE PRINT "YES YES YES"	(And this one also)
YES YES YES	
←	

TEST takes one input which must evaluate to "TRUE" or "FALSE". Its effect is to mark a "truth flag" correspondingly (i.e., to true or false). The associated command IF TRUE takes an instruction line as input and executes this line only if the truth flag is marked true. (Similarly with IF FALSE, when the truth flag is marked false.) Note that IF TRUE and IF FALSE take *command lines* as inputs -- in this respect they are different from other LOGO commands.

Using TEST and IF TRUE, new predicates can be defined by LOGO procedures. Thus:

```
+TO NOT /INPUT/
>10 TEST /INPUT/
>20 IF TRUE OUTPUT "FALSE"
>30 OUTPUT "TRUE"
>END
NOT DEFINED
+PRINT NOT IS "1" "2"
TRUE
←
```

2.9 Recursion

A procedure may use another procedure in its definition, as we saw in Section 2.6 where QUADRUBBLE was defined using DUBBLE. An even more powerful capability comes about by using a procedure in its own definition. An example of such a self-referential, or *recursive*, procedure is the following one, FIND, which outputs the /N/th element of /LIST/.

```
+TO FIND /LIST/ /N/
>10 TEST IS /N/ 1
>20 IF TRUE OUTPUT (FIRST OF /LIST/)
>30 OUTPUT FIND OF (BUTFIRST OF /LIST/)
    AND (DIFFERENCE OF /N/ AND 1)
>END

+PRINT FIND OF "Z Y X W" AND 1
Z
+PRINT FIND OF "Z Y X W" AND 3
X
←
```

FIND outputs immediately only in the case where /N/ is 1. It reduces all other cases to that one by creating a number of distinct copies of FIND. For the last example shown, three

separate copies of FIND were used for execution of that instruction. The process was as follows:

The command line is:

```
PRINT FIND "Z Y X W" 3
```

PRINT needs one input, so the execution goes to the next element on the line, which is FIND. Two literal inputs follow FIND, so FIND is executed line by line. Line 30 of FIND with the current inputs is

```
30 OUTPUT FIND "Y X W" 2.
```

Thus, in order to finish execution of our original procedure FIND, we must perform FIND "Y X W" 2. A new copy of FIND, let us denote it (for ourselves) as FIND*, is used for this purpose and this copy is given inputs "Y X W" and 2. When line 30 of FIND* is reached, we have

```
30 OUTPUT FIND "X W" 1.
```

Thus, still another copy of FIND is required, which we call FIND**, with inputs "X W" and 1. The execution of FIND** results in an output of "X" to FIND* (since the value of /N/ for this copy is 1) and FIND** "vanishes". FIND* now outputs "X" to FIND, which outputs "X" to PRINT.

We can display this sequence of successive procedure calls with the inputs and output associated with each of them in a compact way using the LOGO command TRACE. We indicate that we wish to TRACE the procedure FIND in its subsequent executions as follows.

```
+TRACE FIND
```

```
+
```

(LOGO puts a trace on FIND and returns control to the user)

The effect of TRACE is illustrated next, using the example just discussed.

```
+PRINT FIND OF "Z Y X W" AND 3
```

```
FIND OF "Z Y X W" AND "3"
```

```
  FIND OF "Y X W" AND "2"
```

```
    FIND OF "X W" AND "1"
```

```
      FIND OUTPUTS "X"
```

```
    FIND OUTPUTS "X"
```

```
  FIND OUTPUTS "X"
```

```
X
```

```
+
```

(This is our FIND*)

(Our FIND**, the third copy of FIND)

(FIND** outputs to FIND*)

(FIND* outputs to FIND)

(FIND outputs to PRINT)

(PRINT prints "X")

Note that TRACE prints the title line of each TRACEd procedure invoked, listing the inputs it is called with, and note that it prints a new line each time a procedure outputs or ends. The title line and output line of each procedure are indented the same number of spaces.

In the procedure FIND the execution of the command OUTPUT requires invoking and executing another copy of the procedure. In some recursive procedures the execution of other operations in the recursion line may be deferred as well. One such procedure is COLLAPSE, which takes a sentence and collapses it into a word.

```

←TO COLLAPSE /S/
>1Ø TEST IS /S/ /EMPTY/ (/EMPTY/ denotes the empty LOGO object, "")
>2Ø IF TRUE OUTPUT /EMPTY/
>3Ø OUTPUT WORD OF (FIRST OF /S/) AND
      (COLLAPSE OF BUTFIRST OF /S/)
>END

```

```

←PRINT COLLAPSE OF "CAN YOU READ ME"
CANYOUREADME

```

In the recursion line of COLLAPSE, line 3Ø, the execution of the operation WORD, as well as the command OUTPUT, must be deferred. This is shown in the following trace.

```

←TRACE COLLAPSE
←PRINT COLLAPSE OF "MARES EAT OATS"
COLLAPSE OF "MARES EAT OATS"
  COLLAPSE OF "EAT OATS"
    COLLAPSE OF "OATS"
      COLLAPSE OF ""
        COLLAPSE OUTPUTS ""
        COLLAPSE OUTPUTS "OATS"
        COLLAPSE OUTPUTS "EATOATS"
        COLLAPSE OUTPUTS "MARESEATOATS"
MARESEATOATS
←

```

The same form of recursion used with COLLAPSE was shown in Section 1 in the definitions of FACTORIAL and REVERSE. More complex and powerful forms of recursion can be created by the advanced user. These can even include recursions which are not reducible to iteration, such as the Ackerman function (the generalized exponential function used in recursive function theory).

3. Program Manipulation

Up to now we have studied only those parts of the LOGO language necessary for writing executable LOGO programs (the operations, commands, names, etc., and the rules governing their relations and usage). This chapter deals with those facilities of LOGO that aid a user in his programming work at the computer terminal. These include listing, editing, erasing, abbreviating, storing, and retrieving.

3.1 Editing

A. Editing a Line

There are a number of ways to modify the instruction line being typed in, at any time before the carriage return key is pressed. This "editing" capability works with both direct instruction lines and those which are part of a procedure definition.

If the backslash character "\" is pressed, this character is typed and its effect is to erase the character preceding it. The backslash can be typed more than once to effect multiple erasures. Thus:

```
+PINT \ \ RINT 4 \ 5
5
+PRIN T4 \ \ T 4
4
+
```

Pressing the CTRL key and W simultaneously results in a number of backslashes being typed, sufficient to completely erase the preceding LOGO word. Denoting this action by W^C , then:

```
+PRINT "THE QUIKWC \ \ \ QUICK BROWN FOX"
THE QUICK BROWN FOX
+
```

The more drastic action of pressing the RUBOUT key erases the entire line. To show this has been done, the computer erases out the back arrow (+) preceding the "rubbed out" line, using a #. Denoting this editing action by RUBOUT, then:

```
+PRINT "THE QUICK BROOW (RUBOUT)
+
```

B. Editing a Procedure

There are several editing commands which can be used only while a procedure is being defined. (The computer indicates that we are in the process of defining a procedure by typing a ">" rather than a "+" when it is ready to receive the next line.) The command EDIT, followed by a procedure name, is used to modify the definition of a previously defined procedure. LOGO responds to this command by typing a ">" to indicate that we are again in defining mode. After the desired changes have been made, the command END terminates the procedure definition just as before.

Inserting a Line

LOGO arranges the lines of a procedure in order of increasing line number. Thus, we can "insert" an instruction line between two already typed lines simply by giving it a number between the line numbers of the two given lines. It is good programming practice to number procedure lines by fives or by tens, to leave space for this possibility.

Assume we had previously defined REV:

```
TO REV /A/ /B/ /C/
10 PRINT /C/
20 PRINT /A/
END
```

and we wish to insert the instruction line PRINT /B/ between lines 10 and 20, as line 15, say. Then:

```
<EDIT REV
>15 PRINT /B/
>END
REV DEFINED
<
```

This effects the desired insertion.

Changing an Entire Line

We can change a previously entered instruction line by simply typing the desired instruction line, giving it the same number. The first version vanishes. We can retype the title line by typing TITLE, followed by the new title line.

Typing a line number and carriage return results in that line containing no instruction -- effectively erasing the line previously having that line number. A neater way to erase a line is by using the command ERASE LINE __, which completely expunges the line indicated.

The current version of any line can be shown by the LOGO command LIST LINE __, which prints out the indicated line. Similarly, LIST TITLE prints the title line.

Changing Part of a Line

Often it is easier to modify an existing line than to completely retype it. To do this, we type EDIT LINE __, giving the appropriate line number. The computer will place the line number at the beginning of the next line. To have it type the next word of that line, we press the control key (CTRL) and N simultaneously. (This action is denoted by N^C.) To get the rest of the line, we type R^C (the control key and R). These two actions, together with "\" and W^C described above, can be used together, as in the following example:

```
>LIST LINE 30      (Note that we are already in defining mode,
30 PRONT SUM OF AND 2  the only context in which LIST LINE and
>EDIT LINE 30      EDIT LINE are meaningful)
30 NCPRONTWC\\\\"PRINT NC SUM NCOF 3 RC AND 2 (carriage return)
```

(Since N^C and R^C don't type out anything on the teletype, the above line looks readable.)

```
>LIST LINE 30
PRINT SUM OF 3 AND 2
```

Listing and Erasing the Entire Procedure

LIST (procedure name) results in the procedure being typed exactly as it stands at that moment. ERASE (procedure name) results in the procedure being expunged.

Nearly any LOGO instruction line including LIST (procedure name) and ERASE (procedure name) can be executed when in the procedure definition (>) mode. The only exceptions are those like TO and EDIT which involve the definition of yet another procedure while we are already defining one.

The following example shows a typical editing session involving the use of most of the features described in this section.

```

+LIST REVERSE
TO REVERSE /Y/ (Should be /X/ in place of /Y/)
10 TEST IS /X/ /EMPTY/ (Missing is 15 IF TRUE OUTPUT /EMPTY/)
20 OUTPUT WORD (LAST /X/)
    (REVERS BUTLAST /X/) (Incorrect spelling)
END
+EDIT REVERSE (We could have used EDIT TITLE and then
>TITLE TO REVERSE /X/ NC twice to have LOGO type TITLE TO
>15 IF TRUE OUTPUT /EMPTY/ REVERSE)
>EDIT LINE 20
20 NCOUTPUT NCWORD NC(LAST NC/X/) NC
    (REVERS \E RCBUTLAST /X/) (The \ deletes the space)
>LIST LINE 20
20 OUTPUT WORD (LAST /X/ (REVERSE
    BUTLAST /X/)
>LIST REVERSE
    (skips one line)
TO REVERSE /X/
10 TEST IS /X/ /EMPTY/
15 IF TRUE OUTPUT /EMPTY/
20 OUTPUT WORD (LAST /X/) (REVERSE
    BUTLAST /X/)
>END (Note there is no END command since
REVERSE DEFINED definition of reverse is not complete)
+

```

3.2 Abbreviating

To reduce the user's typing, the computer recognizes short forms for most commands. These are called abbreviations. For example:

```

+P S "CAT" "DOG"
CAT DOG

```

P is the abbreviation for PRINT and S for SENTENCE. The long forms are substituted internally for the abbreviations as soon as the abbreviations are typed in. Thus, if we type in a procedure definition making use of abbreviations and then list it, the computer types it back to us in expanded form. The set of built-in abbreviations is given as part of Section 5.

The user can make his own abbreviations with the command ABBREVIATE (two inputs). The first input can be any LOGO command, operation, or procedure, or combination of them. The second is the word which will become the abbreviation. The "noise word" AS may be inserted between the two inputs of ABBREVIATE.

```

+ABBREVIATE "PRINT SUM" AS "+"
++ "3" "5"
8
+

```

An abbreviation can refer to just one operation. If we type:

```

+ABBREVIATE "RANDOM" "R"
+ABBREVIATE "REVERSE" "R"

```

the first meaning of "R" is lost.

Built-in abbreviations can also be changed.

+ABBREVIATE "POWER" "P" causes the abbreviation P for PRINT to vanish.

Listing and Erasing Abbreviations

LIST ALL ABBREVIATIONS results in the typing of all user-defined abbreviations, for example, we now have:

```

+LIST ALL ABBREVIATIONS

```

```

R:  REVERSE
P:  POWER
+:  PRINT SUM
+

```

ERASE ALL ABBREVIATIONS is used to erase all abbreviations, and ERASE ABBREVIATION (abbreviation) to erase the indicated abbreviation. For example,

```

+ERASE ABBREVIATION "+"
++ 2 3
+ IS UNDEFINED
+

```

3.3 The User Workspace

Upon logging in on a computer and requesting LOGO, the user has at his disposal all built-in features of the LOGO language. These include the LOGO operations and commands, reserved names, such as /EMPTY/, and standard abbreviations. The user is also assigned a *workspace* within the computer memory. The additions he makes to the LOGO built-ins are kept in this workspace. These possible

additions include user-defined procedures, user-defined abbreviations, and those user-defined names which were not created by execution of procedure title lines. Each class of objects in the user workspace can be listed or erased separately:

```
LIST ALL PROCEDURES
ERASE ALL PROCEDURES
```

```
LIST ALL NAMES
ERASE ALL NAMES
```

```
LIST ALL ABBREVIATIONS
ERASE ALL ABBREVIATIONS
```

Abbreviations are the only built-ins which can be changed by the user. ERASE ALL ABBREVIATIONS not only erases all user abbreviations, but restores the built-in abbreviations to their original state.

The commands LIST ALL and ERASE ALL combine the listing and erasing commands for all these three types of objects in user workspace. Thus, LIST ALL provides an exact accounting of everything in the user workspace and ERASE ALL completely empties the workspace. Any procedure, name, or abbreviation can be listed or erased individually, as described in preceding sections.

The command LIST CONTENTS lists just the title line of every procedure in workspace. For example:

```
+LIST CONTENTS
```

```
TO REVERSE /X/
TO FACTORIAL /N/
+
```

GOODBYE, as well as exiting from LOGO, results in the complete loss of the user workspace. LOGO provides commands to save the contents of the workspace for subsequent use. If such retention is desired, it must be effected before GOODBYE is typed. The saved material can then be retrieved at any later time. This process of "SAVEing" and "GETting" is described next.

3.4 Filing

LOGO provides a facility for users to file away their work. The basic unit of a LOGO file is an *entry*. Each entry has a two-word *entry name*. The first word of the entry name is the *file name* and is common to all the entries in a file (it is commonly the

name of the user who owns the file). The second word of the entry name distinguishes the entry from other entries in the same file. Examples of entry names are JIM EQUATIONS, NANCY RANDOMSENT, JIM NIM, NANCY EQUATIONS.

An entry is created by the command SAVE. The entry thus created contains everything in the user workspace -- that is, everything that would be listed by LIST ALL. The user workspace is left unchanged by the SAVE command. For example,

```
+SAVE GRANT ARITH
+
```

In this example, the entry GRANT ARITH is created. If GRANT ARITH already exists, the old entry is replaced by the new one. Although the user workspace may subsequently change, the contents of the entry GRANT ARITH will remain as they were at the time they were saved. When the user gives the command GOODBYE, the workspace is destroyed but all entries are retained.

In a well-organized file, each entry contains a related group of procedures, names, and abbreviations (for example, those that are used for playing NIM, or those used in solving linear equations). By first erasing irrelevant parts, the user can save any desired subset of his workspace.

To retrieve an entry from a file, the command GET is used.

```
+GET GRANT ARITH
+
```

The contents are copied into, and become a part of, the student's current workspace -- the entry itself is unchanged. The additions to the workspace provided by a GET are inserted in the same way as if a user had typed them in. Thus, abbreviations and names supersede existing ones and procedures in the entry having the same names as those in the workspace are not entered.

There are three different types of objects possible in the user workspace, and hence in any entry. These parts of an entry can be listed by the commands

```
LIST PROCEDURES _____
LIST NAMES      _____
LIST ABBREVIATIONS _____
```

(where the dashes
indicate the two-word
entry name)

LIST ENTRY _____ lists everything in the entry -- all three parts. LIST CONTENTS _____ gives the title line of each procedure contained in the entry indicated.

LISTing any part of an entry does not result in its being copied into the active workspace. Only GET will do this.

The command LIST ALL FILES causes LOGO to type all existing file names. The command LIST FILE _____ causes LOGO to type the entries in a given file. To remove an entry from a file, the command ERASE ENTRY is used:

```
+ERASE ENTRY GRANT ARITH
+GET GRANT ARITH
THERE IS NO ENTRY GRANT ARITH
+
```

When all entries in a file are erased, the file itself is automatically eliminated.

LOGO also provides two *operations* for working with LOGO files.

SIZE _____ outputs a number proportional to the amount of space the entry indicated occupies in memory.

ENTRIES _____ outputs a sentence of the second words of all entry names contained in the file indicated.

3.5 Debugging

The LOGO system has built-in aids to help users find the "bugs" in their programs. A bug has one of two effects. It may cause the computer to try to execute an illegal instruction or it may direct the execution of instructions that are legal but which produce a wrong answer or no answer at all, e.g., it may put the computer in a loop that never ends.

In the first case, the computer immediately stops executing instructions and types out a diagnostic message describing the error and telling where it occurred. (Some typical diagnostic messages are listed at the end of this section.) An example of this is:

```
+TO GREET /X/
>10 PRINT SENTENCE OF "HELLO," AND /X/
>20 PRONT "HOW ARE YOU?"
>30 PRINT "SEE YOU LATER"
>END
GREET DEFINED
```

```
+GREET "JOHN"  
HELLO, JOHN
```

```
PRONT NEEDS A MEANING.  
I WAS AT LINE 20 IN GREET
```

There was a bug. The diagnostic message designates the type of error and where the error was found. EDIT can be used to make the necessary changes.

```
+EDIT GREET  
>20 PRINT "HOW ARE YOU?"  
>END  
GREET DEFINED
```

```
+GREET "JOHN"  
HELLO, JOHN  
HOW ARE YOU?  
SEE YOU LATER  
+
```

When the procedure GREET was being defined, the computer didn't object when line 20 was typed in, even though it did not know the meaning of PRONT. The reason is that a procedure PRONT might have been written later, after GREET was defined but before it was executed.

In this example, the computer's diagnostic message pointed to the source of the error and thus was directly helpful. Often, however, we get situations where the illegal instruction isn't the direct cause of the error. For example, in the course of running a procedure the computer may say

```
DIFFERENCE OF "AB" AND "1"? INPUTS MUST BE NUMBERS.  
I WAS AT LINE 30 OF SAM.
```

```
+EDIT SAM  
>LIST LINE 30  
30 OUTPUT SUM OF /X/ AND PRODUCT OF /X/ AND DIFFERENCE OF  
/Y/ AND "1"
```

Assuming the arithmetic expression given is the one intended, the error is not contained in line 30. Somewhere earlier in the execution, /Y/ was made "AB" instead of a number. This type of error then is of the second type mentioned above. The computer gets past the faulty instruction and the defective result shows up as a bug later when the computer is performing another instruction, perhaps in a different procedure. In this situation the diagnostic is less helpful and the error is more difficult to track down.

The TRACE command, described in the section on recursion, is often useful in finding errors, especially those resulting in a faulty recursion. TRACE, followed by a procedure name, results in a special "flag" being placed with the procedure. Then, whenever the procedure is called in an execution, its title line is typed with the current values of its dummy variables. When the procedure outputs, or is otherwise completed, an appropriate typeout is made and the output, if any, is shown. If a procedure is in TRACE mode, this is indicated whenever the procedure is LISTed. To get out of TRACE mode, the command ERASE TRACE, followed by the procedure name, is used. ERASE ALL TRACES is a more drastic LOGO command.

Another useful approach, when the difficulty lies within a known procedure rather than "between" procedures, is to insert extra lines in the defective procedure to type intermediate results. These help to pinpoint the error and can be removed after a correction has been made.

Diagnostic Messages

There are about 100 diagnostic messages. The following are some typical ones.

YOU NEED / MARKS AROUND EACH INPUT.
TITLE MUST BE FOLLOWED BY "TO".
END WHAT? YOU'RE NOT DEFINING ANYTHING.
GO WHERE?
LIST ALL WHAT?
DON'T TRY TO DEFINE ANOTHER PROCEDURE INSIDE THIS ONE.
DIVISION BY ZERO.
DON'T USE THE EMPTY WORD FOR A NAME.
ERASE WHAT?
THE INPUT TO TEST MUST BE A PREDICATE.

The following comments mean that the number of inputs found on the line was not correct. The exact form of comment depends on the particular parsing error.

 IS EXTRA
THERE ARE INPUTS MISSING FOR

In the following diagnostics, the underscored words are filled in appropriately by LOGO when the error occurs. The words given here are typical examples.

MATCHING"? (or / or (or))
PRONT NEEDS A MEANING.
TRUMP ISN'T COMPLETELY DEFINED. (END command not yet given.)

THERE IS NO LINE 30.

SUM OF "A" AND "5"? INPUTS MUST BE NUMBERS.

TEST IS USED BY LOGO.

(The user cannot define a procedure called TEST)

OF ISN'T A PROCEDURE.

THERE ISN'T ANY FILE GRANT

REVERSE IS ALREADY DEFINED.

YOU'RE ALREADY DEFINING REVERSE.

YOU'RE ALREADY EDITING REVERSE.

REVERSE CAN'T BE USED AS AN INPUT. IT DOESN'T OUTPUT.

The comment I AM IN TROUBLE. TELL YOUR TEACHER indicates a computer failure.

3.6 Interrupting Execution

The execution of a direct line or procedure is interrupted by the momentary depression of the key labeled BREAK. The pressing of the BREAK key is effective, whether the computer is performing internal operations or printing on the teletypewriter. When this occurs, LOGO types "BREAK" as well as the procedure name and line it was then executing, and then returns control to the user. (This "positional" information is omitted if a direct line was interrupted.) The state of the execution is preserved -- all intermediate results are kept. These include all the local names set up by the use of dummy variables in procedures which had not yet terminated. LIST ALL NAMES gives all these "local" names in order opposite to the order of their creation. Control has returned to the user exactly as though these intermediate results did not exist. They do not get saved by a SAVE command, nor do they interfere with any procedure definition or execution. They can, however, slow down execution somewhat because they take up room in the user's workspace. The only real effect these intermediate results have is initiated by the no-input command GO. This results in the interrupted calculation being resumed exactly from the point left off. The only loss that can occur is that of some printing that was in process when BREAK was pressed. Any changes made in the interim will, of course, result in a continuation different from that produced if no interruption had taken place. Thus, the BREAK key can be a useful debugging tool.

The no-input command CANCEL erases the intermediate results produced by the calculation interrupted by the latest BREAK. Thus, if an execution has been interrupted, another initiated without the use of GO and the BREAK key again pressed, two uses of CANCEL are needed to erase all the intermediate results existing in the workspace. It is good practice to use CANCEL after interrupting any procedure which is not to be resumed using GO.

4.1 Formatting

TYPE is a command which causes its one input to be typed. It differs from PRINT only in that the typehead remains positioned just after the character last typed -- there is no return to the beginning of the next line.

/BLANK/	a blank space. (the space bar cannot be used for this because LOGO eliminates superfluous spaces.)
/LINE FEED/	teletype goes to next line without moving horizontally.
/CARRIAGE RETURN/	return to beginning of current line.
/SKIP/	a new line. (has the effect of both a carriage return and a line feed.)
/FORM FEED/	move paper to new page on teletypes which have a form feed feature.

```

←TO DIAGONAL /N/
>10 TEST IS /N/ 0
>20 IF TRUE STOP
>30 TYPE "X"
>40 TYPE /BLANK/
>50 TYPE /LINE FEED/
>60 DIAGONAL (DIFF /N/ 1)
>END
DIAGONAL DEFINED
←DIAGONAL 8

```

4.2 Interactive Programs

All commands and operations used thus far must have their inputs specified before they are executed. Each such input is a literal, the thing of a name, or the output of some operation or procedure. The LOGO operation REQUEST, however, causes execution to pause until the user has typed a string of characters and a carriage return. REQUEST then outputs this string. For example:

```
+PRINT REQUEST
*HUMBUG                      (REQUEST prints an asterisk "*" to show
HUMBUG                      that user type-in is required*)
+PRINT SUM OF REQUEST AND REQUEST
*3                          (The leftmost REQUEST)
*2
5
+
```

REQUEST makes possible the writing of programs which "interact" with the user.

```
+TO COPYCAT
>10 PRINT "TELL ME SOMETHING."
>20 PRINT REQUEST
>30 COPYCAT
>END
COPYCAT DEFINED
+COPYCAT
TELL ME SOMETHING.
*WHO ARE YOU?
WHO ARE YOU?
TELL ME SOMETHING.
*WHY SHOULD I?
WHY SHOULD I?
TELL ME SOMETHING.
*ARE YOU SOME KIND OF NUT
ARE YOU SOME KIND OF NUT
TELL ME SOMETHING.
*
:
:
:
```

* This asterisk is omitted when LOGO is not at the left-hand edge of the paper. This is often the case when the last typing resulted from a TYPE command.

The existence of an interactive capability makes the element of time particularly interesting. There are several ways in which LOGO makes provision for timing.

The operation ASK requires one input which must be a number. ASK is the same as REQUEST unless the user has not completed his typing when a number of seconds equal to the input has elapsed. If this happens, ASK outputs the empty word and returns the carriage to a new line.

```
+TO QUICKQUERY /QUESTION/
>10 PRINT /QUESTION/
>20 MAKE "ANSWER" ASK 5
>30 TEST IS /ANSWER/ /EMPTY/
>40 IF TRUE PRINT "YOU WEREN'T FAST ENOUGH"
>50 OUTPUT /ANSWER/
>END
QUICKQUERY DEFINED
+PRINT QUICKQUERY "WHO DISCOVERED FERMAT'S LAST THEOREM?
  YOU HAVE 5 SECONDS TO ANSWER."
*FERM      (If as here, 5 seconds have elapsed before the user
            presses CARRIAGE RETURN, LOGO resumes control)
YOU WEREN'T FAST ENOUGH
            (The empty line is printed here by QUICKQUERY)
+
```

The current date and time are made available to LOGO by the no-input operations DATE and TIME:

```
+PRINT DATE
6/13/71
+PRINT TIME
11:05 PM
```

There is also an internal "clock" which is started when the user enters LOGO. This clock keeps time in seconds:

```
+PRINT CLOCK
1806
+PRINT CLOCK
1811
```

RESET CLOCK sets the clock back to 0.

```
+PRINT CLOCK
1825
+RESET CLOCK
+PRINT CLOCK
5
+
```

The WAIT command makes LOGO pause a number of seconds equal to its one input. It has no other effect. For auditory interaction, there is the reserved name "BELL". PRINT /BELL/ rings the teletype bell.

4.3 More Arithmetic

There are several numerical operations besides the basic four operations and GREATERP and RANDOM, all discussed earlier. All built-in numerical operations require integer inputs.

The operation QUOTIENT simply outputs the integer part of the quotient of its two inputs. REMAINDER outputs the remainder of the division yielding the quotient. DIVISION outputs a sentence of two numbers -- the quotient of its two inputs and the remainder.

```
+PRINT QUOTIENT OF 34 AND -6
-5
+PRINT REMAINDER OF 34 AND -6
4
+PRINT DIVISION OF 34 AND -6
-5 4
←
```

MAXIMUM outputs the greater of its two inputs. MINIMUM outputs the lesser of its two inputs.

```
+TO ORDER2 /A/ /B/
>10 OUTPUT SENTENCE
    MINIMUM /A/ /B/
    MAXIMUM /A/ /B/
>END
ORDER2 DEFINED
+PRINT ORDER2 3 -1
-1 3
←
```

ZEROP is a one-input operation which outputs "TRUE" or "FALSE" as the input is, or is not, numerically equal to zero. Thus, ZEROP is not the same as IS 0.

```
+PRINT IS 0 00
FALSE
+PRINT ZEROP 00
TRUE
←
```

COUNT is not itself a numerical operation, but it has an integer output, so it usually appears in conjunction with numerical operations. COUNT has one input. Its output is the number of letters in the input, if it is a LOGO word -- or the number of words, if it is a LOGO sentence. Thus:

```
+PRINT COUNT OF "ABC"  
3  
+PRINT COUNT OF "THE CAT IN THE HAT"  
5  
+PRINT COUNT ""  
0  
+
```

4.4 Local and Global Names

The Command LOCAL

As we saw earlier, including a name on the title line of a procedure meant that a special copy of the name would be created each time the procedure was invoked. Each copy disappears when the procedure which created it terminates. This feature is especially useful when a procedure makes copies of itself recursively, like the procedure REVERSE in the section dealing with recursion. When a name does not appear in the title line of the procedure in which it is used, no special copy of the name is made. This is often a useful feature when we use such a *global name* to transfer information from one procedure to another. Sometimes, as in the example following, a "slight" variation of REVERSE, it is a handicap.

```
+TO REVERSE /INPUT/  
>10 TEST IS /INPUT/ /EMPTY/  
>20 IF TRUE OUTPUT /EMPTY/  
>30 MAKE "Y FIRST OF /INPUT/  
>40 OUTPUT WORD OF  
    REVERSE (BUTFIRST OF /INPUT/) AND /Y/  
>END  
REVERSE DEFINED  
+PRINT REVERSE "HELLO"  
00000
```

```

+TRACE REVERSE
+PRINT REVERSE OF "HELLO"
REVERSE OF "HELLO"      (/Y/ is now "H")
  REVERSE OF "ELLO"     (/Y/ is now "E")
    REVERSE OF "LLO"    (/Y/ is now "L")
      REVERSE OF "LO"   (/Y/ is now "L")
        REVERSE OF "O"  (/Y/ is now "O")
          REVERSE OF ""  (Since input is /EMPTY/, /Y/ is
            REVERSE OUTPUTS ""      not changed)
              REVERSE OUTPUTS "O"
                REVERSE OUTPUTS "OO"
                  REVERSE OUTPUTS "OOO"
                    REVERSE OUTPUTS "OOOO"
                      REVERSE OUTPUTS "OOOOO"
                        OOOOO
+

```

(The result is PRINTed)

"Y" is made "O" in the fifth copy of REVERSE and never changes thereafter. In the succeeding outputs, this /Y/ is what is actually used. We really intended that a new copy of "Y" exist for each calling of REVERSE. We can easily accomplish this by the insertion of the instruction:

```
5 LOCAL "Y"
```

This results in "Y" being handled just as if it were on the title line. Now REVERSE works.

```

+PRINT REVERSE "HELLO"
OLLEH
+

```

LOCAL is an unusual command in that it allows any number of inputs to follow it. Each is taken as a name to be made "local" to the procedure in which LOCAL appears.

4.5 Automatic Program Generation - An Advanced Feature

DO is a LOGO command which results in the execution of its one input. Thus, for example,

```

+DO "EDIT FOO"
>

```

(and we are editing FOO)

In the case above we could just as well have typed in EDIT FOO directly, omitting the quotes and the DO. But, we could also have written, in still larger form:

```
+MAKE "PNAME" "FOO"
+DO SENTENCE "EDIT" AND /PNAME/
```

Here we see the utility of DO. It enables exact specification of parts of statements to be deferred, which otherwise would have had to be inserted in literal form. Thus the command DO forms the basis for general procedures which create or modify other procedures.

Two operations are provided by LOGO to enable a procedure to find the current contents of a procedure to be modified.

LINES is a one-input procedure. It outputs the sentence of the line numbers of the procedure given as input.

TEXT is a two-input procedure -- it requires a procedure name and a line number. It outputs the entire line, as a sentence. If the line number 0 is given, TEXT outputs the title line of the procedure indicated.

Thus, LINES can be used to find what lines exist and TEXT to go through them one-by-one.

To illustrate these commands, consider the following procedure for replacing /WORD/ by /SUBST/ in /SENTENCE/.

```
+TO REPLACE /WORD/ /SUBST/ /SENTENCE/
>10 TEST IS /SENTENCE/ /EMPTY/
>20 IF TRUE OUTPUT /EMPTY/
>30 TEST IS (FIRST /SENTENCE/) /WORD/
>40 IF TRUE OUTPUT SENTENCE
      /SUBST/
      REPLACE /WORD/ /SUBST/ (BUTFIRST /SENTENCE/)
>50 OUTPUT SENTENCE
      FIRST /SENTENCE/
      REPLACE /WORD/ /SUBST/ (BUTFIRST /SENTENCE/)
>END
REPLACE DEFINED
+PRINT REPLACE OF "CATS" "DOGS" "IT'S RAINING CATS AND DOGS"
IT'S RAINING DOGS AND DOGS
+PRINT LINES OF "REPLACE"
10 20 30 40 50
+PRINT TEXT OF "REPLACE" AND 30
30 TEST IS (FIRST /SENTENCE/) /WORD/
+
```

The following set of procedures, using the procedure REPLACE, inserts /W2/ in place of /W1/ everywhere the latter appears in a procedure /PNAME/. It exemplifies the use of DO, LINES, and TEXT.

```
TO MODIFY /PNAME/ /W1/ /W2/
10 DO SENTENCE OF "EDIT" AND /PNAME/
20 CHANGE (LINES OF /PNAME/) /W1/ /W2/
30 DO "END"
END
```

```
TO CHANGE /LINES/ /W1/ /W2/
10 TEST IS /LINES/ /EMPTY/
20 IF TRUE STOP
30 DO SENTENCE SENTENCE SENTENCE
   "REPLACE"
   /W1/
   /W2/
   TEXT OF (FIRST /LINES/)
40 CHANGE (BUTFIRST OF /LINES/) /W1/ /W2/
END
```

The reserved name "CONTENTS" has as its thing, the sentence of all procedure names (not title lines) in workspace. This makes possible writing of procedures even more general than the above. For further generality, /FILES/ is a sentence consisting of all file names.

4.6 Other Ways to Terminate a Procedure

To stop execution of a procedure before the END command, the Ø input command STOP may be used. Its effect is exactly that of END -- the procedure simply stops and control returns to whatever called the procedure. STOP is often used to terminate one branch resulting from a TEST. For example,

```
TO FACTOR /A/
10 TEST ZEROP /A/
20 IF TRUE PRINT "I CANNOT FACTOR ZERO"
30 IF TRUE STOP
: : :
```

The one-input command EXIT also terminates a procedure. Its input is typed, then LOGO acts exactly as though an error has been found, typing the location of the EXIT command and returning

control to the user. Thus, the example of a partial procedure preceding could have been written

```
TO FACTOR /A/
10 TEST ZERO /A/
20 IF TRUE EXIT "I CANNOT FACTOR ZERO"
  ::
```

and, after the procedure has been completed,

```
+FACTOR 0
I CANNOT FACTOR ZERO
I WAS AT LINE 20 OF FACTOR
```

4.7 Miscellany

The Turtle

There is a set of operations and commands reserved for the "turtle", a LOGO-controlled robot.

The 0-input commands FRONT and BACK move the turtle one unit in the directions they name. RIGHT and LEFT rotate the turtle clockwise and counterclockwise. HORN rings the turtle's bell.

The 0-input operations TOUCH LEFT and TOUCH RIGHT refer to the turtle's touch sensors. TOUCH LEFT outputs TRUE if the left sensor is against an obstacle, otherwise, it outputs FALSE. TOUCH RIGHT queries the right touch sensor in the same way.

Other Commands and Operations

EMPTY is a one-input operation which outputs TRUE or FALSE as its input is or is not the empty word. EMPTY has exactly the same effect as IS /EMPTY/.

IGNORE is a one-input command which has no effect. It is used in the rare situations where an output which has no further use is generated. For example, the lines

```
50 PRINT "PRESS CARRIAGE RETURN TO CONTINUE"
60 IGNORE REQUEST
```

result in typing of the message given by line 50, followed by typing of "*" and a pause in execution until the CARRIAGE RETURN key is pressed.

GO TO LINE is a one-input command which is valid only in a procedure. It causes execution to pass to the line whose number is given as its input.

Comments

A user may place remarks which he does not wish to be executed anywhere within a procedure definition. The user indicates that a string is not to be executed by placing semicolons around it.* The only restriction on inserting comments in this way is that they may not be placed within the quotes demarcating a literal or within the pair of slashes delimiting a LOGO name. Remarks correctly indicated have no effect on the execution of the procedure they lie within. They only appear when the procedure is listed. For example,

```
TO DIAGONAL /N/; DRAWS A DIAGONAL LINE*
1Ø TEST IS /N/ Ø; END TEST
2Ø ; IF /N/ IS Ø WE ARE DONE; IF TRUE STOP
3Ø TYPE "x"
4Ø TYPE /BLANK/; MOVE ACROSS ONE SPACE
5Ø TYPE /LINE FEED/; NEXT LINE
6Ø ; REPEAT FOR /N/ -1; DIAGONAL (DIFF /N/ 1)
END
```

Comments may be placed after the entry name in a SAVE command, again preceded by a semicolon. The comment is typed whenever the entry name appears in a listing:

```
+SAVE GRANT ARITH; A GENERAL ARITHMETIC PACKAGE
```

/QUOTE/

If the user tries to print "DOG", complete with quotes, he gets an error.

```
+PRINT ""DOG""
      (prints empty word)
DOG"" IS EXTRA
```

The special LOGO thing /QUOTE/ is used in such a situation to indicate a quote mark which is not intended as the delimiter of a literal. Thus:

```
+PRINT SENTENCE SENTENCE /QUOTE/ "DOG" /QUOTE/
"DOG"
```

*A comment at the end of a line need not be terminated with a semicolon.

5. Glossary and Index

	Abbr.	Description	Page
ABBREVIATE	ABT	(2-input command) sets up second input as the abbreviation of the first input	23
AND		("noise" word) used for clarity; valid only between inputs of a procedure	6
AS		("noise" word) valid only in abbreviating	23
ASK		(1-input operation) outputs literal type-in from teletype if completed in input number of seconds, else the empty word	33
BACK		(0-input command) "turtle" effector, moves turtle back 1 space	39
BOTH	B	(2-input operation) each input must be TRUE or FALSE; outputs TRUE if both are TRUE, otherwise FALSE	16
BUTFIRST	BF	(1-input operation) outputs all but first character of an input word, or all but first word of input sentence	7
BUTLAST	BL	(1-input operation) outputs all but last character of input word, or all but last word of input sentence	7
CANCEL		(0-input command) eliminates one level of break	30
CLOCK		(0-input operation) outputs time given by internal one-second clock	33
COUNT	C	(1-input operation) outputs number of characters of an input word, or number of words of an input sentence	35
DATE		(0-input operation) outputs current date	33
DIFFERENCE	DIFF	(2-input operation) difference of first and second input, which must be integers	8
DIVISION	DIV	(2-input operation) inputs must be integers. Output is sentence of integer quotient and remainder	34
DO		(1-input command) executes its input as a LOGO instruction line	36
EDIT		(command followed by procedure name) puts LOGO into define mode	21, 23, 28
EDIT LINE	EDL	(1-input command) used to edit line indicated, valid only in define mode	22, 23

	Abbr.	Description	Page
EDIT TITLE	EDT	(Ø-input command) used to edit title, valid only in define mode	23
EITHER	EI	(2-input operation) each input must be TRUE or FALSE. Outputs TRUE if either input is TRUE, otherwise FALSE	16
EMPTY	EP	(1-input operation) outputs TRUE if input is the empty word, otherwise FALSE	39
END		(Ø-input command) terminates a procedure definition	5
ENTRIES		(1-input operation) outputs sentence of second words of entry names in file given as input	27
ERASE	ER	(command, followed by procedure name) erases procedure from workspace	22
ERASE ABBREVIATION		(1-input command) erases abbreviation given as input	24
ERASE ALL		(Ø-input command) completely erases workspace, restores built-in abbreviations	25
ERASE ALL ABBREVIATIONS		(Ø-input command) erases all user-defined abbreviations in workspace	24,25
ERASE ALL NAMES		(Ø-input command) erases all user-defined names in workspace	25
ERASE ALL PROCEDURES		(Ø-input command) erases all procedures from workspace	25
ERASE ALL TRACES		(Ø-input command) removes trace flag from all traced procedures in workspace	29
ERASE ENTRY	EE	(command, followed by entry name) completely erases indicated entry	27
ERASE LINE	ERL	(1-input command) only valid in define mode. Erases line whose number is given as input	21
ERASE TRACE		(command, followed by procedure name) removes trace flag from indicated procedure	29
EXIT		(1-input command) types its input, the line number and procedure name in which it appears and terminates execution	38
FIRST	F	(1-input operation) outputs the first character of an input word, or the first word of an input sentence	7

	Abbr.	Description	Page
FRONT		(Ø-input command) "turtle" effector, moves turtle forward 1 space	39
GET		(command, followed by entry name) enters indicated entry into workspace	26
GO		(Ø-input command) continues execution from a BREAK key interrupt	30
GO TO LINE	GTL	(1-input command) only valid within a procedure definition. Transfers execution to line whose number is given as input	40
GOODBYE	GB	(Ø-input command) terminates LOGO session	3, 25
GREATERP	GP	(2-input operation) inputs must be integers. Outputs TRUE if first input is strictly greater than second, else FALSE	16
HORN		(Ø-input command) "turtle" effector, rings turtle's bell	39
IF FALSE	IFF	(command followed by instruction) executes instruction if truth flag is FALSE, otherwise has no effect	16
IF TRUE	IFT	(command followed by instruction) executes instruction if truth flag is TRUE, otherwise has no effect	16
IGNORE		(1-input command) has no effect	39
IS		(2-input operation) outputs TRUE if first input is identical to second, otherwise FALSE	16
LAST	L	(1-input operation) outputs last character of an input word, or last word of input sentence	7
LEFT		(Ø-input command) "turtle" effector, rotates turtle counterclockwise	39
LINES		(1-input operation) output is sentence of all instruction line numbers of procedure whose name is given as input	37
LIST		(command followed by procedure name) types definition of indicated procedure	22
LIST ABBREVIATIONS		(command followed by entry name) types all abbreviations in entry indicated	26
LIST ALL		(Ø-input command) types entire user workspace	25

	Abbr.	Description	Page
LIST ALL ABBREVIATIONS		(Ø-input command) types all abbreviations in user workspace	24
LIST ALL FILES		(Ø-input command) types all file names	27
LIST ALL NAMES		(Ø-input command) types all names in user workspace	25,30
LIST ALL PROCEDURES		(Ø-input command) types definitions of all procedures in user workspace	25
LIST CONTENTS	LC	(Ø-input command) types title lines of all procedures in user workspace	25
LIST CONTENTS	LC	(command followed by entry name) types title lines of all procedures in entry indicated	27
LIST ENTRY	LE	(command followed by entry name) types entire entry indicated	27
LIST FILE		(command followed by file name) types list of entries in file indicated	27
LIST LINE	LL	(1-input command) valid only in define mode. Types line whose number is given as input	22,23
LIST NAMES		(command followed by entry name) types all names in entry indicated	26
LIST PROCEDURES		(command followed by entry name) types all procedure definitions in entry indicated	26
LIST TITLE		(Ø-input command) valid only in define mode. Types title line of procedure being defined	22
LOCAL		(command with any number of inputs) only valid in a procedure. Makes all names given as inputs local to the procedure containing this command	35
MAKE		(2-input command) makes the first input the name of the second input	12
MAXIMUM	MAX	(2-input operation) inputs must be integers. Outputs the greater of the inputs	34
MINIMUM	MIN	(2-input operation) inputs must be integers. Outputs the lesser of the inputs	34
NUMBERP	NP	(1-input operation) outputs TRUE if input is an integer, otherwise FALSE	15

	Abbr.	Description	Page
OF		("noise" word) used for clarity; only valid following procedure name	6
OUTPUT	OP	(1-input command) valid only in procedure. Passes its input to procedure (or operation or command) which called current procedure	11
PRINT	P	(1-input command) types its input on teletype and returns carriage to beginning of next line	3
PRODUCT	PROD	(2-input operation) inputs must be integers. Outputs their product	6,8
QUOTIENT	QUO	(2-input operation) inputs must be integers. Outputs their integer quotient	8,34
RANDOM		(Ø-input operation) outputs a random digit	8
REMAINDER	REM	(2-input operation) inputs must be integers. Outputs remainder of their integer division	34
REQUEST	RQ	(Ø-input operation) outputs literal type-in from teletype	32
RESET CLOCK		(Ø-input command) resets internal clock to zero	33
RIGHT		(Ø-input command) "turtle" effector, rotates turtle clockwise	39
SAVE		(command followed by entry name) saves user workspace as entry with name indicated	26
SENTENCE	S	(2-input operation) outputs sentence of its inputs	7
SENTENCEP	SP	(1-input operation) outputs TRUE if input is a sentence, otherwise FALSE	15
SIZE		(1-input operation) outputs "size" of entry given as input	27
STOP		(Ø-input command) valid only in procedure. Terminates execution of its procedure	38
SUM		(2-input operation) inputs must be integers. Outputs their sum	8
TEST	T	(1-input command) input must be TRUE or FALSE. Sets truth flag to its input	16

	Abbr.	Description	Page
TEXT		(2-input operation) outputs text in procedure given by first input, with line number given by second input	37
THING		(1-input operation) outputs thing named by the input	12
TIME		(0-input operation) outputs current time	33
TITLE		(command followed by title line) valid only in define mode. Changes title of procedure being defined to that following	21
TO		(command followed by title line) enters define mode of procedure name following	4
TOUCH LEFT		(0-input operation) "turtle" feedback, outputs TRUE if left sensor has touched obstacle, resets touch flag, otherwise outputs FALSE	39
TOUCH RIGHT		(0-input operation) "turtle" feedback, outputs TRUE if right sensor has touched obstacle, resets touch flag, otherwise outputs FALSE	39
TRACE		(command followed by procedure name) sets trace flag for procedure indicated	18, 29, 36
TYPE		(1-input command) types its input	31
WAIT		(1-input command) causes execution to pause a number of seconds equal to its input	34
WORD	W	(2-input operation) inputs must be words. Outputs the word formed by concatenating them	7
WORDP	WP	(1-input operation) outputs TRUE if input is a word, otherwise FALSE	15
ZEROP	ZP	(1-input operation) outputs TRUE if input is equal to zero, otherwise FALSE	34

There are also several abbreviations for parts of commands.
They are

ABB: ABBREVIATION
 ABBS: ABBREVIATIONS
 ER: ERASE
 PRS: PROCEDURES

Reserved Names

	Description	Page
/EMPTY/	the empty thing	19
/CONTENTS/	a sentence of user-defined procedure names	38
/LINE FEED/	a line feed without carriage return when typed	31
/CARRIAGE RETURN/	a carriage return without line feed when typed	31
/FILES/	a sentence of file names	38
/FORM FEED/	moves paper to a new page when typed on teletypes having form feed feature	31
/BLANK/	a blank space when typed	31
/BELL/	rings a bell when typed	34
/QUOTE/	a quote mark	40
/SKIP/	a new line (carriage return and line feed) when typed	31

Special Keys

RETURN	gives line just typed by user to LOGO	3
RUB OUT	erases line being typed	3,20
LINE FEED	carriage goes to next line without terminating current line	3
BREAK	interrupts execution	3,30
\	deletes last character typed	20
*CTRL-W	deletes last word typed	20
*CTRL-N	only valid in EDIT LINE mode. Gets next word of line being edited	22
*CTRL-R	only valid in EDIT LINE mode. Gets rest of line being edited	22
*CTRL-B	same as /BLANK/	
*CTRL-G	same as /BELL/	
*CTRL-L	same as /FORM FEED/	

*this notation indicates that the CTRL key is pressed simultaneously with indicated letter.

Special Characters

	Description	Page
"	used to delimit literals	4
/	used to delimit names	5,13
()	"noise" word, used for clarity	6,11
;	used to delimit comments	40
←	typed by LOGO to indicate user control	3
>	typed by LOGO to indicate define mode	5,22
#	typed by LOGO at beginning of line to indicate line has been "RUBbed OUT"	20
*	typed by LOGO to indicate REQUEST needs user type-in	32